Aalto University
School of Science

Olavi Pesonen

REAL TIME LOG LENGTH MEASUREMENT USING GPU ACCELERATED VISUAL ODOMETRY

Thesis for the degree of Licentiate of Science in Technology submitted for inspection, Espoo, 18 August, 2015.

Supervising professor Tapani Raiko

Thesis advisor

Aalto University
School of Science

Lisensiaatintutkimuksen tiivistelmä

Tekijä

Pesonen Olavi

Lisensiaatintutkimuksen nimi

REAL TIME LOG LENGTH MEASUREMENT USING GPU ACCELERATED VISUAL ODOMETRY

Tiivistelmä

Tämän lisensiaatintutkimuksen aiheena on GPU laskennan käyttö konenäköön perustuvassa tukin pituuden mittauksessa. Konenäköön perustuva pituuden mittaus ei tarvitse uudelleen kalibrointia puulajin tai lämpötilan mukaan. Konenäköön perustuvassa mittauksessa myöskään mittapyörä ei voi luistaa tukin pinnalla.

Realiaikaisuuden vaatimus on tässä sovelluksessa korkea. Kuvat on otettu 120 Hz taajuudella, koska leikkuupää liikuttaa tukkia useita metrejä sekunnissa. GPU laskenta potentiaalisesti nopeuttaisi laskentaa tarvittavissa määrin. Realiaikaista vastetta haettiin sekä algoritmien valinnalla että harkitsemalla mahdollisuuksia rinnaisohjelmoinnin käyttämiseen. Monessa tapauksessa vasteet paranivat, vaikka grafiikkakortin ominaisuudet usein rajoittivat rinnaikkaisohjelmoinnista saatavaa hyötyä.

Realiaikainen vaste saavutettiin, mutta ei tarvittavalla pituuden mittaamisen tarkkuudella. Molempien tavoitteiden saavuttaminen jäi mahdollisten jatkotöiden tehtäväksi.

Tutkimusala **Informaatioteknologia (T101Z)**	**Avainsanat** **Konenäkö, stereonäkö, tukin pituuden mittaus**
Vastuuprofessori **Apulaisprofessori Tapani Raiko**	**Sivumäärä** **70**
Ohjaaja	**Kieli** **Englanti**
Työn tarkastaja **TkT Heikki Huttunen**	**Päiväys** **18.8.2015**

Luettavissa verkossa osoitteessa https://aaltodoc.aalto.fi/handle/123456789/27

Aalto University
School of Science

Abstract of Licentiate Thesis

Author

Pesonen Olavi

Title of Thesis

REAL TIME LOG LENGTH MEASUREMENT USING GPU ACCELERATED VISUAL ODOMETRY

Abstract

This thesis studies GPU accelerated visual odometry in measuring log length. The visual odometry would not suffer slippage nor require recalibration depending type of wood or temperature conditions compared to mechanical measurement.

The requirement of the real-time performance is quite high. Image capturing in 120 Hz frequency is needed as log is moved several meters per second by harvester heads. Here GPU acceleration will be used as it can give speedup in magnitude of hundreds or more. Real-time performance is targeted by selecting fast algorithms for subtasks of measurement pipeline and considering possibilities to parallelize algorithm. In many cases performance boost is achieved, but not in expected magnitude. Physical constraints of the graphics card hardware become easily the limiting factor in parallelization.

Real-time performance was achieved in this thesis but not with required accuracy. It remained for future work to find out which algorithms would give both targets.

Research field **Information technology (T101Z)**	**Keywords** **Visual odometry, log length measurement, stereo vision**
Supervising professor **Assistant Professor Tapani Raiko**	**Pages** **70**
Thesis advisor	**Language** **English**
Thesis examiner **Dr.Eng. Heikki Huttunen**	**Date** **18.8.2015**

The thesis can be read at https://aaltodoc.aalto.fi/handle/123456789/27

Acknowledgements

I present thanks to professor's Erkki Oja, Arto Visala and Tapani Raiko, who made this thesis possible. Main part of my studies were taken in Erkki Oja's department. He also recommended for me when applying for graduate studies. Subject of this thesis was found in conversations with Arto Visala. Tapani Raiko acted as supervising professor in finalizing this thesis work.

I also thank authors of previous thesis in this same subject: M.Sc. Jakke Kulovesi and M.Sc. Jouko Kalmari. My work was based on their effort.

Table of Contents

List of Abbreviation and Symbols

$\boldsymbol{X}$	(uppercase bold) matrices
$\boldsymbol{x}$	(lowercase bold) vectors
${}^{w}\boldsymbol{x}$	coordinates of vector **x** are given in (w)orld frame
${}^{c}\boldsymbol{P}_{w}$	transformation matrix from (w)orld to (c)amera frame
$[\boldsymbol{t}]_x = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$	Skew symmetric matrix
CPU	central processing unit
GPU	graphics processing unit
CUDA	Compute Unified Device Architecture
ILP	Instruction Level Parallelism
SLAM	simultaneous localization and mapping
SM	streaming multiprocessor of Fermi architecture
SMX	streaming multiprocessor of Kepler architecture
TLP	Thread Level Parallelism

1 Introduction

In Finland mechanical harvesting is economically more feasible option than manual harvesting in most locations. Currently almost all the fellings are done mechanically by harvesters[1]. Manual harvesting is preferred choice in locations where damage to terrain should be kept minimal like in parks or near buildings.

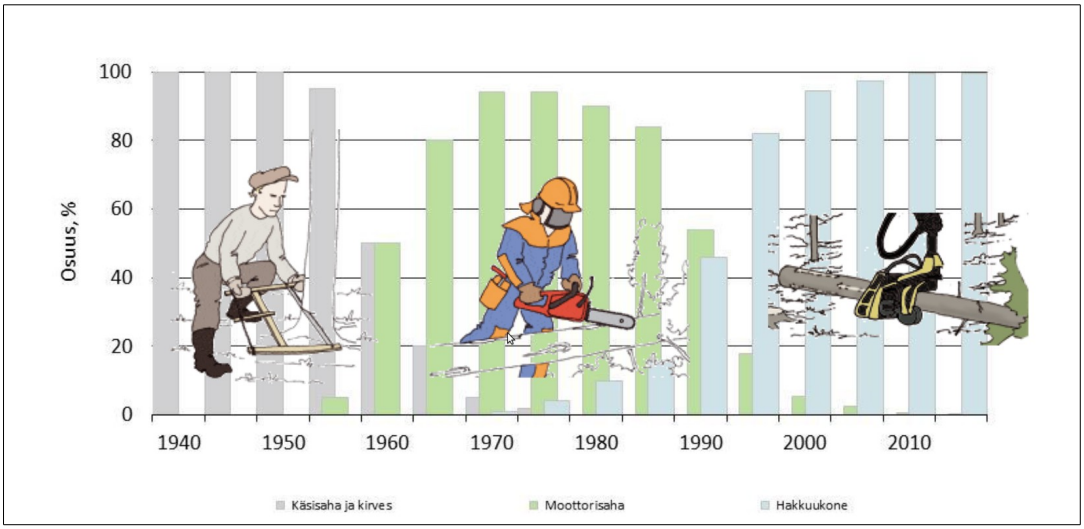


Illustration 1: Felling techniques from 1940's to today [1].

The mechanical harvesting has two major methods: tree-length and cut-to-length methods. In the tree-length method, trunks are delimbed in the forest and transported to the mill whole or almost whole. In the cut-to-length method, the trunks are cut in the forest by their intended use and then transported to the mill.

In Scandinavia cut-to-length is the dominating harvesting method.

The cut-to-length method has following advantages

- lower operating costs as fewer machines are needed
- simpler logistics as the logs are cut to correct lengths by the intended usage
- reduced logging damage to terrain as the the machines can be relatively light compared to machines handling whole trunks
- capability to pick special wood
- smaller patches become economically feasible
- mechanized selective thinning becomes possible

Flip side of the coin is higher investing cost. The harvesters capable to cut-to-length are technically more advanced than the tree-length forest machines and cost more. The harvesters contain instruments and software to measure diameter and length of log. Otherwise cutting and sorting by the intended use would not be possible at all.

1.1 Measuring

Trade in wood is based on a measured amount of the wood: either volume or weight. Final price of the deal is calculated by the measured amount of the wood. Both involved parties are interested that the measured volume is correct.

Wood can be measured by the harvester or at at the road side or at the mill. Also measurement by the weight scale is possible. These four measuring categories are currently mostly used.

Image below shows how much each method of the measurements were used in the year 2012 in total commercial round-wood fellings [2]. Clearly the measurement by the harvester is mostly used. Fellings in industry and state owned forests use more the measurement at the mill. However, in privately owned forests the measurement by the harvester dominates. The volume of private owned fellings was 30.8 million m^3 and the volume of industry and the state owned fellings was 11.6 millions m^3. This results in the shares that are shown in the chart below.

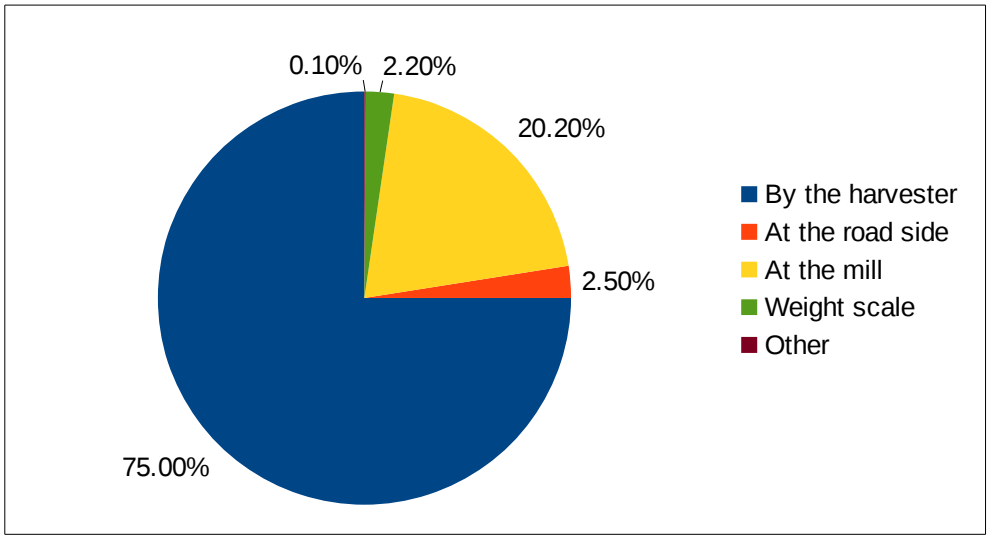


Illustration 2: The measuring techniques used in Finland in the year 2012.

Factories use commonly a laser technology and are very precise. Pulse encoders attached to the log feeder are also used to measure log length. However at the factory the logs are processed after measuring: either cut into pulp or milled to paper. Verification is not possible if seller doubts reported amounts.

The log measuring has been considered worth of legislation in Finland. Finnish law requires that maximum deviation is 4% by the harvester head.

1.2 Describing harvester head.

Cut to length operation is an essential functionality of the harvester head. The head measures the length of the log passed through and displays measurement to the driver. The driver cuts the log to the desired length.

Currently the log length is measured by measuring wheel or by the feeder depending on the manufacturer. In the image 3 is an example of the measuring wheel. It can be faintly seen in the middle. It is in principle a cogwheel, the rolling of which is measured.

Illustration 3: Measurement wheel is located in the middle of the harvester head, pointed by red arrow.

Circumstances have effect on the measurement by the running wheel. Type of the wood affects how deep the teeth cut into the log. This results in an incorrect length. Currently the operator manually enters type of wood into the harvester. Also temperature affects the measurement as frozen wood is harder. Also varying thickness of bark make the measuring by the measurement wheel unreliable.

As a result measurement wheel needs frequent manual calibration. Daily calibration is part of normal harvester operation. However temperature changes may cause multiple calibrations during working day. In the calibration process the operator manually measures the trunk length and this measured length is entered to harvester.

At best the harvesters can measure the log length with the tolerance of 1 cm. In practice 95% of the trunks fall into ±3 cm window if the length measurement is calibrated as instructed. The diameter is measured with the tolerance of 1 mm.

Removing the need of the calibration would slightly increase productivity of the harvester and improve operator safety.

1.3 Development of stereo vision

Optical measurement has arisen as one alternative that does not suffer these drawbacks. Optical measurement is not sensitive to hardness of wood, thickness of the bark nor temperature conditions. However, optical measurements has it own challenges where performance is one.

Recent development in stereo vision related algorithms has shown applications where SLAM can achieve 20-30 Hz frame rate [3].

1.4 Focus and scope

This thesis studies if parallel computing can be used to achieve real-time performance in optical measurements of log length. For real-time operation frame rate of 120 Hz would be needed. This is a challenge as systems operating at frame rates of 20-30 Hz are referenced as real-time, which is adequate for motion pictures.

Earlier work in this subject has proved that log length can be accurately measured optically. However implementations has been too slow. Real-time performance has not been achieved. In this work GPU acceleration is used for increased performance. Results will show if real-time operation can be achieved.

2 Algorithms for stereo vision

This chapter describes methods and algorithms needed for an optical measurement. In principle this is a structure from motion problem. In the classical setting camera is moving in a stationary scene. In the problem of this thesis camera is stationary and movement of the log is computed by algorithms.

2.1 Projective geometry

Projective geometry describes objects as they appear compared to Euclidean geometry that describes objects as they are. Lengths, angles, parallelism become distorted when we look at the objects. Projective geometry describes mathematical models how the images of the Euclidean 3D world are formed.

Mathematical models are needed when operating on the appearance of the objects in computers. This chapter contains basic building blocks that are needed when dealing with the computer vision. Either creating a 2D image from the 3D world scene or creating a 3D object from the 2D images.

2.1.1 Homogeneous coordinates

Homogeneous coordinates are system of coordinates used in projective geometry like Cartesian coordinates are used in Euclidean geometry, [4] and [5]. In other words homogeneous coordinates are used when dealing with the appearance of the objects while Cartesian coordinates are used when dealing with the objects as they are.

Homogeneous coordinates are ubiquitous in computer graphics and they are used in computer vision as well. This is no surprise as computer vision is essentially same problem but other way round. Homogeneous coordinates are covered in many textbooks, I have found [6] and [7] helpful.

Homogeneous coordinates can be converted from Cartesian coordinates by inserting additional element into vector. Table below shows transformations in 2D and 3D cases.

2D image	3D scene or world
$(x, y) \rightarrow (x, y, 1)$	$(x, y, z) \rightarrow (x, y, z, 1)$

Table 1: Conversion from Euclidean to homogeneous coordinates.

Conversion from homogeneous coordinates to Cartesian is executed by normalizing with the last element and dropping it. Table below shows conversions in 2D and 3D spaces.

2D image	3D scene or world
$(x, y, w) \rightarrow (\frac{x}{w}, \frac{y}{w})$	$(x, y, z, w) \rightarrow (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$

Table 2: Conversion from Homogeneous to Euclidean coordinates.

Higher dimensional homogeneous coordinates are possible but 2D and 3D spaces are most often used in computer vision and graphics.

What is achieved by this conversion? Homogeneous coordinates make projection from 3D to 2D plane linear operation which can be implemented by matrix multiplication. Studying image 4 it can be seen that projection from 3D point to 2D point involves dividing by z-coordinate.

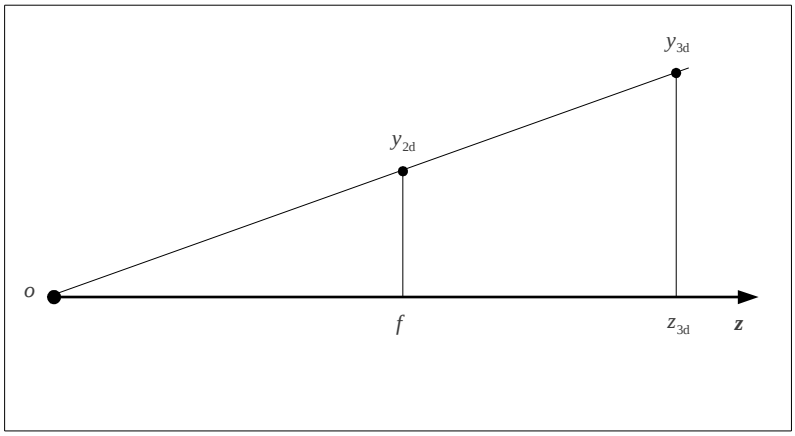


Illustration 4: Calculating image coordinates of the 3D point.

Using the formula for similar triangles we get

$$y_{2d} = f \frac{y_{3d}}{z_{3d}}$$

Same applies also along x-axis. Thus projection from the world to image planes is not linear because division is not a linear operation. But in homogeneous coordinates division is done in normalization as a post processing step.

Consider a point in Euclidean space. Same point in homogeneous coordinates is represented by (*xw*, *yw*, *zw*, *w*). Projecting to 2D plane means dropping last element and normalize by third element. So point (*x*, *y*, *z*) becomes (*x*/*z*, *y*/*z*, 1) and finally dropping now the superfluous *z* coordinate, this becomes (*x*/*z*, *y*/*z*) in plane. In other words projection from 3D top 2D is represented by matrix below in homogeneous coordinates. Normalization comes later.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Matrices representing other geometric transformations can be combined with this and each other by matrix multiplication. As a result, any perspective projection of space can be represented as a single matrix.

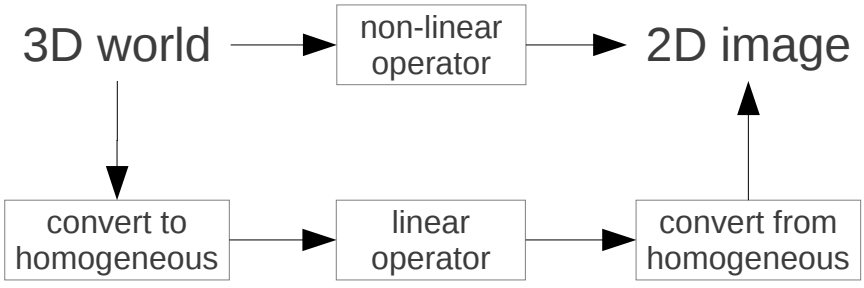


Illustration 5: Conversion to homogeneous coordinates allows usage of linear operator.

Another advantage of homogeneous coordinates is capturing concept of infinity. Homogeneous coordinates where last element is zero are by definition at infinity. And indeed – dividing other elements by zero would place them at infinity. Division by zero is avoided by manipulating coordinates in homogeneous system.

Other linear matrix operations can be applied on homogeneous vectors. These include translation, rotation and scaling.

2.1.2 Projective plane

This subchapter covers topics related to 2D projective plane. Point is represented by triple <wx,wy,w> as mentioned in previous chapter. Line is also presented by triple <a,b,c> where elements come from familiar equation of 2D lines.

$$ax+by+c=0$$

Using matrix notation this equation becomes

$$[a\,b\,c][x\,y\,1]^T=0\Leftrightarrow \boldsymbol{l}^T\boldsymbol{x}=0$$

Here it can be seen that the roles of line and point can be interchanged. Equation above still holds if the roles of the line and the point are interchanged. Neither does multiplying by constant has an effect. This also means that for any theorem that applies to projective plane, there is another theorem where the roles of the lines and the points are interchanged. This is called duality principle.

Equation above is also condition for incidence. Point lies on line if $\boldsymbol{l}^T\boldsymbol{x}=0$ holds. Or a condition for a line to go through a given point.

Other frequently used equations are: line defined by two points and point defined by intersection of two lines. These two equations are good example of the duality principle.

$$\boldsymbol{p}=\boldsymbol{l}_1\times\boldsymbol{l}_2$$
$$\boldsymbol{l}=\boldsymbol{p}_1\times\boldsymbol{p}_2$$

These equations can be used even if the lines are parallel. Two parallel lines meet at infinity but that is not problem in homogeneous coordinates. In Euclidean plane this would not be possible as it would result in division by zero. Additionally points that lie at infinity can used to define a line. Remember homogeneous points with last element set to zero are by definition in infinity.

2.1.3 Projective Space

This chapter covers topics related to 3D projective space. Point is presented by quadruple <wx, wy, wz, w> analogous to projective plane. In projective space quadruple <a,b,c,d> doubles as plane. Elements are coefficiens of familiar equation of plane.

$$ax+by+cz+d=0$$

Using matrix notation this equation becomes

$$[a\,b\,c\,d][x\,y\,z\,1]^T=0\Leftrightarrow \boldsymbol{n}^T\boldsymbol{p}=0$$

In the case of projective space the roles of the plane and the point can be interchanged. That is equation can be written as $\boldsymbol{p}^T\boldsymbol{n}=0$. Neither multiplying by a constant does not have an effect. Duality principle applies to theorems regarding projective space. For any theorem concerning projective space, there is another theorem where roles of points and planes are interchanged.

Equation above is also condition for incidence. A point lies on plane if $\boldsymbol{n}^T \boldsymbol{p} = 0$ holds. Or plane contains a given point.

A 3D line is defined by two points or by intersection of two planes. This is trickier to represent in projective space than single point or plane. A number of representations has been proposed for this purpose. This thesis uses Plücker matrix to represent a 3D line.

Given two points **A** and **B**, the Plücker matrix $\boldsymbol{L}$ representing line going through these points is given by

$$\boldsymbol{L} = \boldsymbol{A}\boldsymbol{B}^T - \boldsymbol{B}\boldsymbol{A}^T$$

For example

$$\boldsymbol{A} = [3\,4\,5\,1]^T$$
$$\boldsymbol{B} = [2\,3\,6\,1]^T$$
$$\boldsymbol{L} = \begin{bmatrix} 0 & 1 & 8 & 1 \\ -1 & 0 & 9 & 1 \\ -8 & -9 & 0 & -1 \\ -1 & -1 & 1 & 0 \end{bmatrix}$$

A Plücker matrix is skew symmetric matrix with rank of 2. Image of $\boldsymbol{L}$ gives the line spanned by **A** and **B**. And null-space of of $\boldsymbol{L}$ is pencil of planes with the line as axis.

Using Plücker matrices intersection of plane and line becomes

$$\boldsymbol{p} = \boldsymbol{L}\,\pi$$

A 3D line could be also defined by intersection of two planes. Given two plane $\boldsymbol{P}$ and $\boldsymbol{Q}$, the Plücker matrix representing line defined by intersection of these planes is given by

$$\boldsymbol{L}^* = \boldsymbol{P}\boldsymbol{Q}^T - \boldsymbol{Q}\boldsymbol{P}^T$$

Plane defined by the join of point and line $\boldsymbol{L}^*$

$$\pi = \boldsymbol{L}^* \boldsymbol{p}$$

Six free parameters in skew symmetric Plücker matrix are caller Plücker coordinates. Line can be defined by giving these coordinates. Plücker coordinates for example above are

$$\Gamma = \{1,8,1,9,1,-1\}$$

2.1.4 Homographies

In projective geometry a homography is bijection of the vector spaces. In computer vision dimension of vector spaces is most often 2 or 3. That is bijections between projective planes or spaces. Higher dimensional homographies are rarely used in the field of computer vision.

Homographies maps lines to lines. In the case of projective plane this means that collinearity, order of contact, intersections and tangency are preserved. In projective space intersection and tangency of planes are preserved.

Homographies can be represented by matrix which operates on homogeneous coordinates. Relationship between linear algebra and projective geometry has been studied in [8].

The property of homographies that lines are mapped to lines, gives a synonym of collineation. Other possible synonyms are projective transform or projectivity. This reveals that perspective transform is a homography.

2.2 Camera model

Camera model is basic building block for computer vision projects. Pinhole camera model is widely used and accepted in computer vision community. Thus it is explained in many computer vision text books, for example by Forsyth and Ponce [9].

Pinhole camera model is demonstrated in image 6. In the pinhole camera model all the rays pass through same point (pinhole), **O.** Image is constructed by tracing rays from objects to image plane through this point. In image 6 there are only two objects (arrows) to be viewed onto image plane.

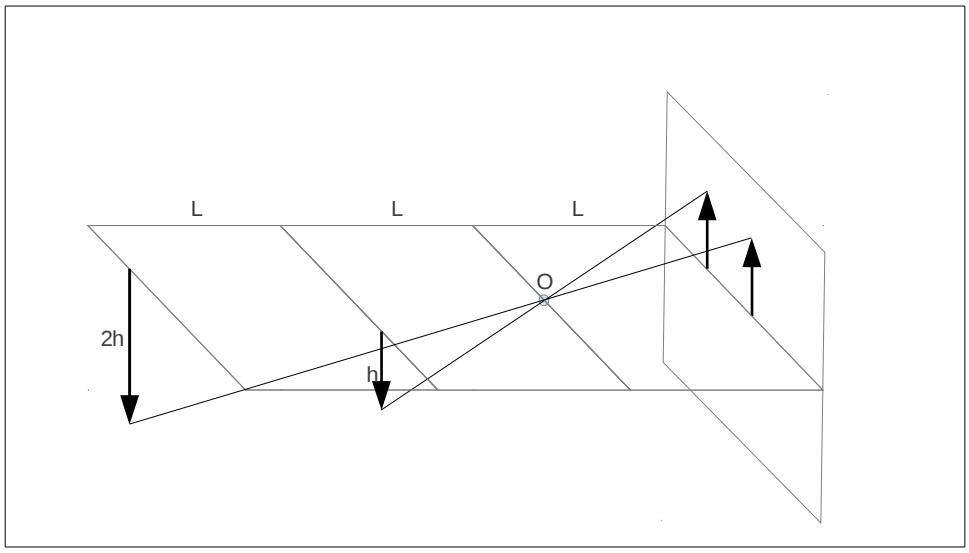


Illustration 6: Projections in pinhole camera.

Image 6 also demonstrates problem associated with single camera. Objects do have same projected height although they differ in real height. Finding real height would require some additional data: image taken by another camera or movement of camera or objects with known lengths.

Formulating pinhole camera model mathematically gives equation below. Here left superscript denotes frame of reference. Thus ${}^{c}\boldsymbol{x}$ means vector presented in coordinate system of the (c)amera.

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Leftrightarrow {}^{s}\boldsymbol{x} = \boldsymbol{K} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} {}^{c}\boldsymbol{x}$$

Here both formula for similar triangles and homogeneous projection is used. In most cases origo is placed in center of image. This is achieved by values of c_x and c_y , which move center of image. The matrix $\boldsymbol{K}$ is contains following intrisic parameters of the camera: focal length *f* and center of image c_x and c_y . The center of image is often referred to as principal point.

Camera calibration matrix $\boldsymbol{K}$ can take other forms too. Most general form is below where focal lengths along x- and y-axis can be freely set. Same number of parameters can be obtained by using aspect ratio α along with focal length. More over skew calibration value *s* can also be set.

$$\boldsymbol{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} f & s & c_x \\ 0 & \alpha f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

2.2.1 Camera extrinsics.

Regarding single image it can be assumed that scene origo is in the world origo. In general this does not hold and taking into account objects position in world coordinate system camera model changes into following format.

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} {}^{c}\boldsymbol{R}_{w} & {}^{c}\boldsymbol{O}_{w} \\ \boldsymbol{0} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Leftrightarrow {}^{s}\boldsymbol{x} = \boldsymbol{K} \begin{bmatrix} {}^{c}\boldsymbol{R}_{w} & {}^{c}\boldsymbol{O}_{w} \end{bmatrix} {}^{w}\boldsymbol{x} = \boldsymbol{P}\,{}^{w}\boldsymbol{x}$$

where

$$\boldsymbol{P} = \boldsymbol{K} \begin{bmatrix} {}^{c}\boldsymbol{R}_{w} & {}^{c}\boldsymbol{O}_{w} \end{bmatrix} = \boldsymbol{K} \begin{bmatrix} {}^{c}\boldsymbol{R}_{w} & -{}^{c}\boldsymbol{R}_{w}\,{}^{w}\boldsymbol{O}_{c} \end{bmatrix} = \begin{bmatrix} \boldsymbol{K}\,{}^{c}\boldsymbol{R}_{w} & -\boldsymbol{K}\,{}^{c}\boldsymbol{R}_{w}\,{}^{w}\boldsymbol{O}_{c} \end{bmatrix} = \begin{bmatrix} \boldsymbol{M} - \boldsymbol{M}\,{}^{w}\boldsymbol{O}_{c} \end{bmatrix}$$

Here is used the notation where the frame of the coordinate system is noted by superscript on left of vector. Notation ${}^{w}\boldsymbol{x}$ means that the elements of vector $\boldsymbol{x}$ are expressed in the (w)orld frame. Regarding matrices used for transformation left superscript means the target coordinate system. The right subscript stands for original coordinate system before transformation. Thus ${}^{w}\boldsymbol{R}_{c}$ means rotation from the (c)amera frame to the (w)orld frame.

Matrix $\boldsymbol{P}$ is generally referred as projection matrix and can be composed and understood in multiple ways as shown by the last line. Here matrix for affine transformation is used to transform coordinates in the world system to coordinates in the camera system.

The last column ${}^{c}\boldsymbol{O}_{w}$ means world origo in camera coordinates, which is part of the affine transform. World origo in camera frame can be expressed by help of camera origin in world coordinates ${}^{w}\boldsymbol{O}_{c}$ and by rotating appropriately, that is $-{}^{c}\boldsymbol{R}_{w}\,{}^{w}\boldsymbol{O}_{c}$. Additionally calibration matrix $\boldsymbol{K}$ can be multiplied inside brackets giving matrix denoted $\boldsymbol{M}$ in this work.

There is strong convention for matrix **K** be upper triangular. It gives possibility to extract camera $\boldsymbol{K}$ from projection matrix by using QR-factorization [10]. Decomposition is needed when projection matrix is starting point, for example camera calibration with known calibration image.

2.2.2 Camera distortions

Pinhole camera is simplification of how light travels in real lens. In real lenses there is distortion compared to pinhole camera. Multiple ways has been discovered to model distortion. Distortion model by Brown and Conrady are widely used [11], [12] and [13]. It models radial and tangential distortions.

Radial distortion is corrected by following equation

$$x_{ideal} = x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$
$$y_{ideal} = y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$
$$where\ r^2 = x^2 + y^2$$

Tangential distortion is corrected by equation below

$$x_{ideal} = x + \left[2\,p_1 x\,y + p_2\left(r^2 + 2\,x^2\right)\right]$$
$$y_{ideal} = y + \left[\,p_1\left(r^2 + 2\,y^2\right) + 2\,p_2 x\,y\,\right]$$
$$where\ r^2 = x^2 + y^2$$

Image 7 shows pair of images: left image is raw image taken by camera; right image shows ideal image where distortion is removed.

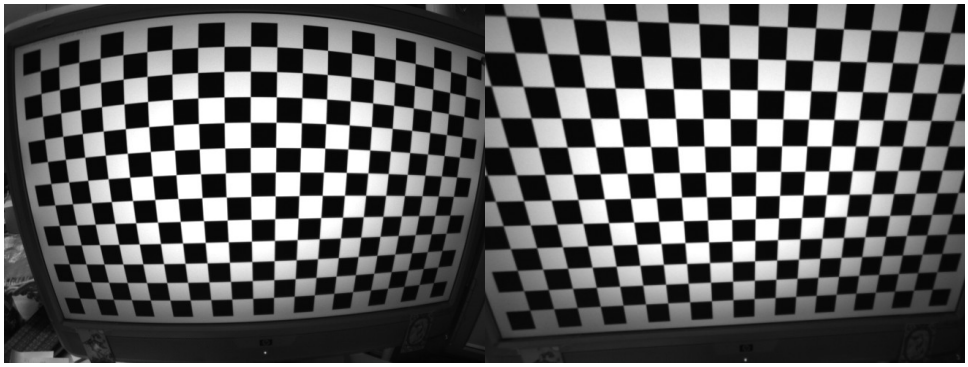


Illustration 7: Example of applying undistortion.

Distortion is often removed by calculating pixel coordinates from ideal to real (distorted) image. Pixels in ideal image can be traversed by integers, one by one. Respective coordinates in distorted image are calculated by formulas above. Coordinates do not coincide with integer indexes in the real image. Some interpolation is used to calculate the value of pixel, e.g. bilinear interpolation.

2.3 Stereo geometry

In the stereo geometry same point $\boldsymbol{x}_{3D}$ is seen in two views. Imposed geometry limits the search space for the corresponding points. This limitation is called the epipolar constraint. It can take form of the essential matrix [14] or the fundamental matrix [15]. The difference here is that essential matrix works in metric units and fundamental matrix in pixels.

2.3.1 Essential matrix

Two cameras viewing same scene gives extra information that can be utilized in finding the corresponding points and ultimately in the 3D construction. Image 8 shows an example, where same point is seen by two cameras. Cameras are commonly noted as left and right camera.

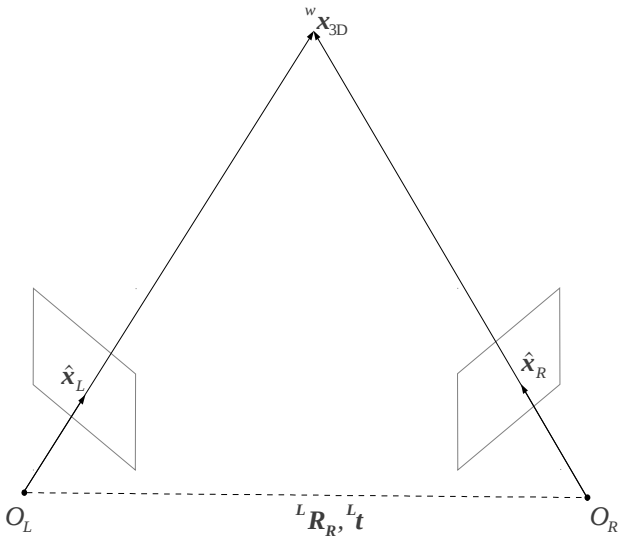


Illustration 8: The epipolar geometry can be defined for two cameras viewing same scene.

Vectors $\hat{x}_L$ and $\hat{x}_R$ are expressed in metric units not in pixels. In other words camera calibration matrix **K** is omitted. These are called normalized camera coordinates. Vectors $\hat{x}_L$ and $\hat{x}_R$ point to the same point ${}^{w}\boldsymbol{x}_{3D}$ which is seen by both the left camera and the right camera. Here again the left superscript denotes frame of the coordinate system. Subscript in vectors are used to distinguish between right and left camera.

$$
\begin{aligned}
{}^{L}\hat{\boldsymbol{x}}_L &= [{}^{L}\boldsymbol{R}_W \quad {}^{L}\boldsymbol{O}_W]\,{}^{W}\boldsymbol{x}_{3D} \\
{}^{R}\hat{\boldsymbol{x}}_R &= [{}^{R}\boldsymbol{R}_W \quad {}^{R}\boldsymbol{O}_W]\,{}^{W}\boldsymbol{x}_{3D}
\end{aligned}
$$

The cameras are separated by the baseline ***t***. Rotation ${}^{L}\boldsymbol{R}_R$ is also needed as cameras point into different directions. Equation below describes the relations between the left and right camera.

$$
\begin{aligned}
&{}^{L}\hat{\boldsymbol{x}}_R = {}^{L}\boldsymbol{R}_R\,{}^{R}\hat{\boldsymbol{x}}_R + {}^{L}\boldsymbol{t}, \textit{where} \\
&{}^{L}\boldsymbol{R}_R = {}^{L}\boldsymbol{R}_W\,{}^{R}\boldsymbol{R}_W^T = {}^{L}\boldsymbol{R}_W\,{}^{W}\boldsymbol{R}_R
\end{aligned}
$$

Points $\boldsymbol{O}_L$, $\boldsymbol{O}_R$ and ${}^{w}\boldsymbol{x}_{3D}$ define plane, which is called epipolar plane. Vectors ${}^{L}\hat{\boldsymbol{x}}_L$, ${}^{L}\boldsymbol{t}$ and ${}^{L}\hat{\boldsymbol{x}}_R - {}^{L}\boldsymbol{t}$ lie on this plane. The essential matrix can be derived by the condition of co-planarity, $\boldsymbol{a}\cdot\boldsymbol{b}\times\boldsymbol{c}=0$. In other words if vector ***a***, ***b*** and ***c*** lie on same plane, take cross product two of them making vector perpendicular to plane. Dot product of this with any vector on plane should be zero. So the derivation of essential matrix becomes

$$
\begin{aligned}
&\begin{cases} \left({}^{L}\hat{\boldsymbol{x}}_R - {}^{L}\boldsymbol{t}\right)^T {}^{L}\boldsymbol{t}\times{}^{L}\hat{\boldsymbol{x}}_l = \boldsymbol{0} \\ {}^{R}\boldsymbol{R}_L^T\,{}^{R}\hat{\boldsymbol{x}}_R = \left({}^{L}\hat{\boldsymbol{x}}_R - {}^{L}\boldsymbol{t}\right) \end{cases} \Rightarrow \\
&\left({}^{R}\boldsymbol{R}_L^T\,{}^{R}\hat{\boldsymbol{x}}_R\right)^T {}^{L}\boldsymbol{t}\times{}^{L}\hat{\boldsymbol{x}}_L = \boldsymbol{0} \Leftrightarrow \\
&{}^{R}\hat{\boldsymbol{x}}_R^T\,{}^{R}\boldsymbol{R}_L\left[{}^{L}\boldsymbol{t}\right]_x\,{}^{L}\hat{\boldsymbol{x}}_L = \boldsymbol{0} \Leftrightarrow \\
&\hat{\boldsymbol{x}}_R^T\boldsymbol{E}\,\hat{\boldsymbol{x}}_L = \boldsymbol{0}, \textit{where} \\
&\boldsymbol{E} = {}^{R}\boldsymbol{R}_L\left[\boldsymbol{t}\right]_x
\end{aligned}
$$

The derivation uses identity that vector cross product is equal to skew symmetric matrix multiplication.

$$
\boldsymbol{a}\times\boldsymbol{b} = [\boldsymbol{a}]_x\boldsymbol{b} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}\boldsymbol{b}
$$

Matrix **E** is called essential matrix and corresponding points in stereo fulfill the this equation. If equation is not met then the match is an outlier. Essential matrix has following property. Transpose of essential matrix applies if roles of cameras are reversed $\hat{\boldsymbol{x}}_L^T\boldsymbol{E}^T\hat{\boldsymbol{x}}_R = 0$.

2.3.2 Fundamental matrix

Essential matrix is defined in normalized camera coordinates. Same relationship holds in image coordinates i.e. in pixels. In this case matrix is called fundamental matrix ***F.*** Equation for fundamental matrix can be derived from equation of essential matrix simply by taking camera calibration matrices into account.

$$\begin{cases} \hat{\boldsymbol{x}}_R^T \boldsymbol{E}\, \hat{\boldsymbol{x}}_L = 0 \\ \boldsymbol{x}_L = \boldsymbol{K}_L \hat{\boldsymbol{x}}_L \\ \boldsymbol{x}_R = \boldsymbol{K}_R \hat{\boldsymbol{x}}_R \end{cases} \Rightarrow$$

$$\left(\boldsymbol{K}_R^{-1} \boldsymbol{x}_R\right)^T \boldsymbol{E}\, \boldsymbol{K}_L^{-1} \boldsymbol{x}_L = \boldsymbol{x}_R^T \boldsymbol{K}_R^{-T} \boldsymbol{E}\, \boldsymbol{K}_L^{-1} \boldsymbol{x}_L = \boldsymbol{x}_R^T \boldsymbol{F}\, \boldsymbol{x}_L = 0,$$

$$\text{where } \boldsymbol{F} = \boldsymbol{K}_R^{-T} \boldsymbol{E}\, \boldsymbol{K}_L^{-1}$$

Again, corresponding points in stereo fulfill this this equation. Fundamental matrix has following property. Transpose of fundamental matrix applies if roles of cameras are reversed $\boldsymbol{x}_L^T \boldsymbol{F}^T \boldsymbol{x}_R = 0$.

2.3.3 Epipolar lines and epipoles

Geometrically epipolar lines are defined as intersection of image plane and epipolar plane, see image 8.

Algebraically epipolar lines are defined by equations below. Fundamental matrix projects a point to a line in another image. Corresponding point is found on this line. This line is called epipolar line. It reduces search of corresponding point from 2D to 1D, along epipolar line.

$$\boldsymbol{x}_R^T \boldsymbol{F}\, \boldsymbol{x}_L = \boldsymbol{x}_R^T \boldsymbol{l} = 0$$

$$\boldsymbol{x}_L^T \boldsymbol{F}^T \boldsymbol{x}_R = \boldsymbol{x}_L^T \boldsymbol{l} = 0$$

Epipole is intersection of image plane and baseline. All epipolar lines go through epipole. Algebraically epipole is null space of fundamental matrix.

Epipolar constraint is building block in structure from motion techniques. Essentially two images in structure from motion technique are treated as if they where taken by stereo rig. Only calibration of stereo rig does not exist. It could not exist as camera movement is not known in advance.

2.3.4 Plane induced homography

Two images of the same plane are related by a homography if the pinhole camera model is assumed. Assume two cameras are viewing points on same plane, image 9. Without loss of generality first camera can be considered $\boldsymbol{P} = [\boldsymbol{I} \quad \boldsymbol{0}]$ and second camera $\boldsymbol{P}' = [\boldsymbol{A} \quad \boldsymbol{a}]$.

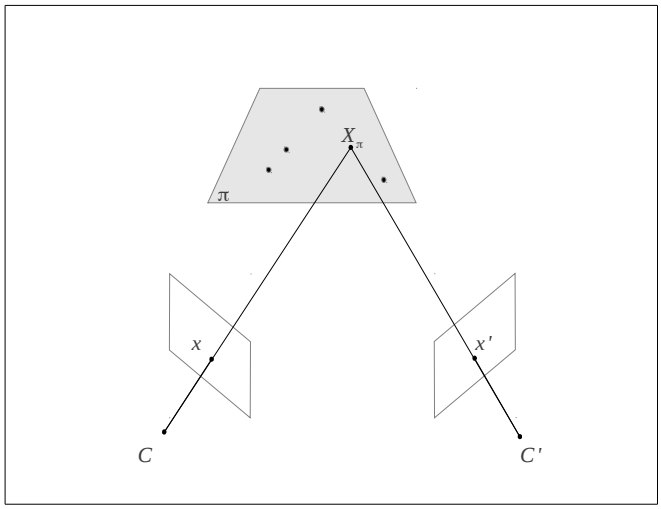


Illustration 9: Plane induce homography can be defined for two cameras viewing same plane.

Using equation for plane and incidence of the plane and point in homogeneous coordinates.

$$\begin{cases} \boldsymbol{\pi}^T \boldsymbol{X} = 0 \\ \boldsymbol{\pi} = [\boldsymbol{v} \;\; 1] \Rightarrow \boldsymbol{X} = [\boldsymbol{x} \;\; -\boldsymbol{v}^T \boldsymbol{x}] \\ \boldsymbol{X} = [\boldsymbol{x} \;\; \rho] \end{cases}$$

Coordinates for $\boldsymbol{X}$ are calculated by finding intersection of given plane $\boldsymbol{\pi}$ and ray going through $\boldsymbol{x}$. This tells $\boldsymbol{X}$ with the help plane normal and point in left camera plane. Now projecting to second image

$$\boldsymbol{x}' = \boldsymbol{P}'\boldsymbol{X} = [\boldsymbol{A} \;\; \boldsymbol{a}]\boldsymbol{X} = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{a}\,v^T \boldsymbol{x} = (\boldsymbol{A} - \boldsymbol{a}\boldsymbol{v}^T)\boldsymbol{x} = \boldsymbol{H}\boldsymbol{x},$$
$$\textit{where } \boldsymbol{H} = \boldsymbol{A} - \boldsymbol{a}\boldsymbol{v}^T$$

In the case of calibrated stereo rig rotation, translation and camera calibration matrices are known. So for the calibrated stereo rig homography becomes.

$$\boldsymbol{\pi} = (\boldsymbol{n}, d) \Rightarrow \boldsymbol{v} = \boldsymbol{n}/d$$
$$\boldsymbol{H} = \boldsymbol{K}_R (\boldsymbol{R} - \boldsymbol{T}\boldsymbol{n}^T/d)\,\boldsymbol{K}_L$$

Homography **H** gives straight dependencies between points in left and right cameras. There is no need to use fundamental or essential matrix to get epipolar line and search for corresponding point.

2.4 Image rectification

Image rectification is a process where captured images are warped so that they can be interpreted as the images taken by the canonized stereo rig [16]. Canonized stereo rig refers to system of two cameras whose optical axis are perfectly aligned to point to same direction. In addition images are rotated so that either x-axis or y-axis of the images coincide. This thesis follows rectification ideas presented in text book by Trucco and Verri [17].

Image 10 presents pair of stereo images before rectification. It can be clearly seen seen that corresponding features do not have same y-coordinates.

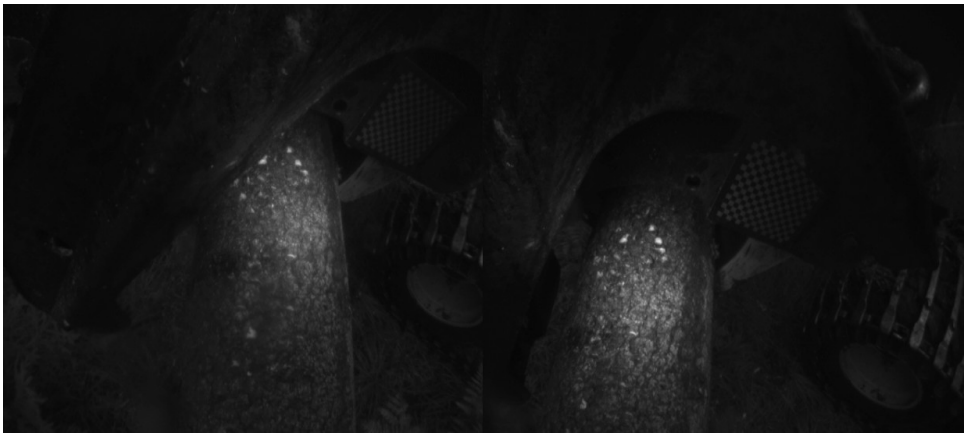


Illustration 10: Example of stereo images before rectification.

Stereo rectification includes finding rotation that aligns x-axis of both images. Such a rotation Q can be found as follows

$$\boldsymbol{q}_1 = \frac{\boldsymbol{T}}{|\boldsymbol{T}|}$$
$$\boldsymbol{q}_2 = \boldsymbol{q}_1 \times \{0, 0, 1\} = \{-T_2, T_1, 0\}$$
$$\boldsymbol{q}_3 = \boldsymbol{q}_1 \times \boldsymbol{q}_2$$

First q_1 is selected to be parallel with translation between cameras. This aligns epipolar lines with x-axis. i.e. epipole is moved to infinity. Vector q_2 is found by taking cross product with optical axis of camera. Finally q_3 is found by cross product again.

Camera intrinsic matrices need to adjusted too. Focal length will be adjusted the same in both cameras. Also principal points are adjusted so that y-coordinates in images match.

Image 11 presents same pair of images as in image 10 but rectified. After rectification corresponding features are aligned by y-axis and have same y-coordinate.

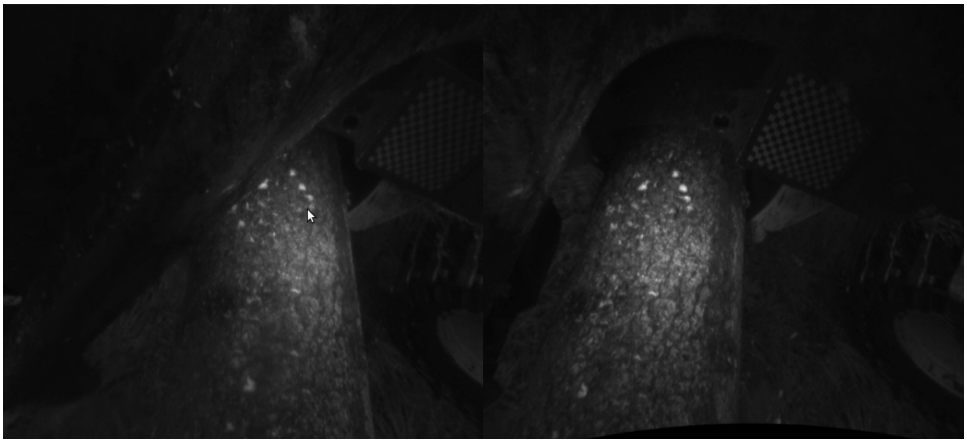


Illustration 11: Example of stereo images after rectification.

In practice rectified images are formed with help of lookup maps. Rectified image needs to be filled pixel by pixel. Steps of traversing occurs on integers, no fractions are included in indexes of the rectified image. Integer location is warped to the raw stereo image and this location might not occur on integer. Pixel intensity stored in the rectified image can be estimated by bilinear interpolation.

2.5 Feature Point Detectors and Descriptors

Detection refers to searching distinctive points in an image. This includes classical corner detection methods like Harris detector or more recent detectors like the SIFT and the SURF. Feature point descriptor refers to calculation of a feature vector. Feature vector is a dimension-reduced representation of the distinctive point.

Heinley et al [18]evaluated performance of binary detector/descriptors and compared them with more traditional detectors and descriptors. Performance evaluation resulted in table below. Some of techniques provide both detector and descriptor. Quite wide range can be seen in performance. In detectors best performance is 2.7 ms while worst is 572 ms. In descriptors best performance is 4.4 microseconds per feature while worst is 314 microseconds per feature.

Detector/descriptor	BRIEF	ORB	BRISK	SURF	SIFT	Harris	MSER	FAST
Detector ms/image	na	17	43	377	572	78	117	2.7
Descriptor us/feature	4.4	4.8	12.9	143	314	na	na	na

Table 3: Performance of interest point detectors and descriptor extractors.

In this work following combination was selected: FAST for detection and BRIEF for descriptor. Looking for performance they were best candidates.

2.5.1 FAST

Feature points in image should be distinguished. It is a prerequisite for correct matches in stereo correspondence or temporal correspondence problem. In stereo match false matches affect triangulation. In the temporal match false matches deteriorates the estimate of the movement.

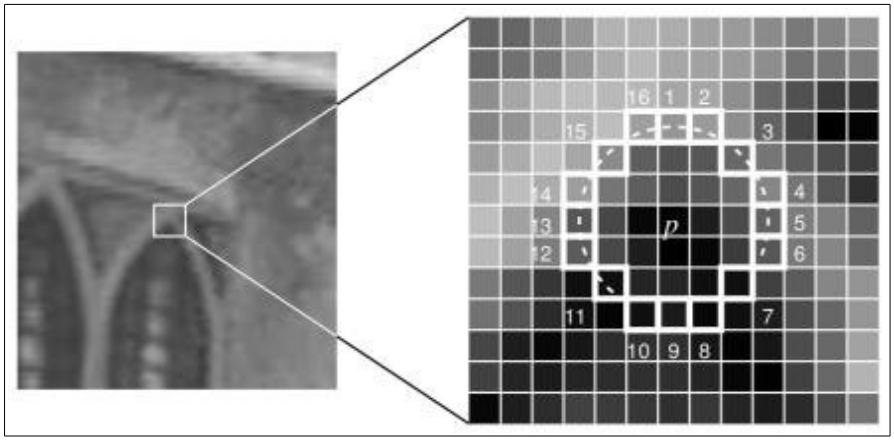


Illustration 12: FAST keypoint detector uses 16 points around tested point [19].

In this work FAST keypoint detector is used, [19] and [20]. FAST relies on studying pixels around interest point I_p . 16 points around the tested point point I_p is considered. If the intensity of 12 or more points is less or more than intensity of the pixel in center then I_p is a keypoint.

Sensitivity of FAST detector can be controlled by adjusting threshold value.

$$I > I_p + threshold \Rightarrow lighter$$
$$I < I_p - threshold \Rightarrow darker$$

FAST includes quick test for rejection. As first step pixel values at locations 1, 5, 9, and 13 are examined. Point I_p can't be interest point if less than three points satisfy threshold condition. In this case full segment scan can be dropped and computation resources saved.

FAST implementations are fast as acronym suggests. FAST is somewhat invariant to lighting conditions too. Change in lighting does not change the relative intensities of the surrounding pixels compared to the center pixel. Pixels darker than the center are still darker than the center. Same applies to the light pixels. However, the threshold has an effect here. Improved lighting rises new keypoints above the threshold and diminishing light reduces the number of detected keypoints.

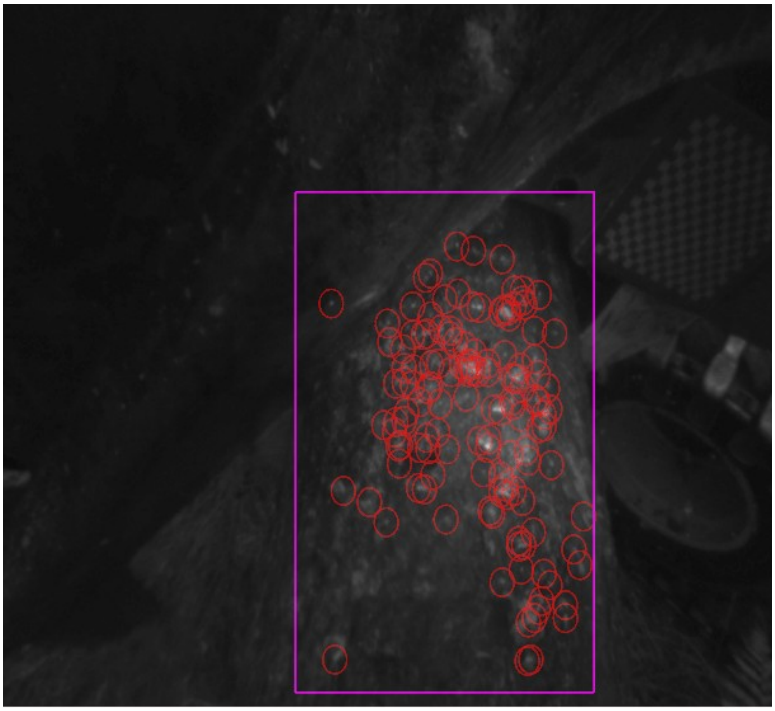


Illustration 13: Example of detected FAST keypoints. Magenta rectangle shows region of interest.

Image 13 shows example of FAST keypoints applied on log frame. Keypoints have been detected on spots where bark has been loosened or where natural texture of bark meets requirements of FAST detection. In this image maximal suppression has been applied too. Only the keypoint with the highest score is reported when keypoints have been detected in adjacent pixels.

2.5.2 BRIEF

Calculating BRIEF (Binary Robust Independent Elementary Features) descriptor is based on pairwise intensity comparisons in small patch around keypoint [21]. OpenCV uses patch size of 48x48 pixels. The descriptor is calculated by comparing intensities on two locations within the patch. Descriptor bit is set to zero or one depending on which location has higher intensity.

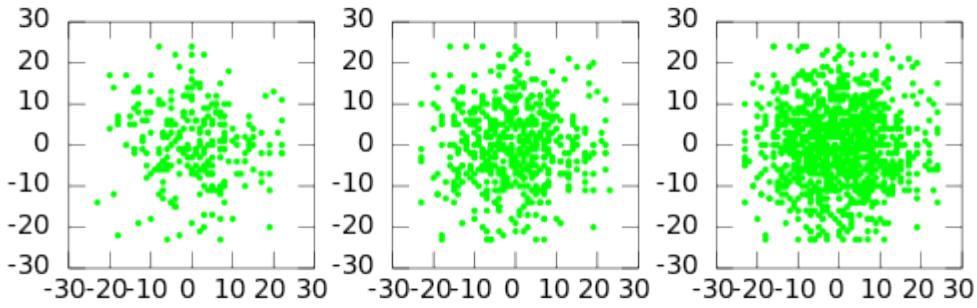


Illustration 14: Locations involved for BRIEF descriptor calculations. From left to right 16, 32 and 64 byte descriptors are illustrated.

Involved locations are randomly sampled from Gaussian distribution. Sampled locations are calculated beforehand so the implementation included hardcoded locations. Possible descriptor lengths are 16, 32 and 64 bytes. In bits corresponding descriptor lengths in bits are 128, 256 and 512 bits. Sampled locations are illustrated in image 14.

Hamming distance is used to find corresponding feature points in given images. BRIEF is not sensitive to changes in lighting. Although illumination changes, relative intensities do not change. Brighter pixels are still brighter and darker pixels are darker compared with each other. However, BRIEF is sensitive to rotations. Hamming distance of descriptor is no longer reliable if clear rotations are involved. Luckily log movement is primarily translation so BRIEF can be used.

BRIEF is simple to calculate and therefore it has small computational impact. BRIEF was found faster than SURF. Moreover it had similar or better recognition performance. This makes it good candidate for real time tracking.

2.6 Non-maximal suppression

Adaptive non-maximal suppression (ANMS) was first introduced by Brown et al. [22]. They used ANMS for image stitching where spatially well distributed interest points produced a more robust transform. In addition it is desirable to restrict the maximum number of interest points extracted from each image as computational cost of matching is super linear. Both of these objectives are useful regarding tracking of log movement.

The idea of ANMS is to find k keypoints with the largest suppression radius r_i where

$$r_i = min_j \|p_i - p_j\| where\, f(p_i) < c_{robust} f(p_j), p_j \in P$$

In other words r_i is distance to nearest stronger keypoint. Keypoint with global maximum strength gets an ∞ radius. Brown et al. used robustification factor $c_{robust} = 0.9$, so keypoint with equal strength does not affect suppression radius. Image 15 shows example of ANMS applied to FAST keypoints detected on log surface.

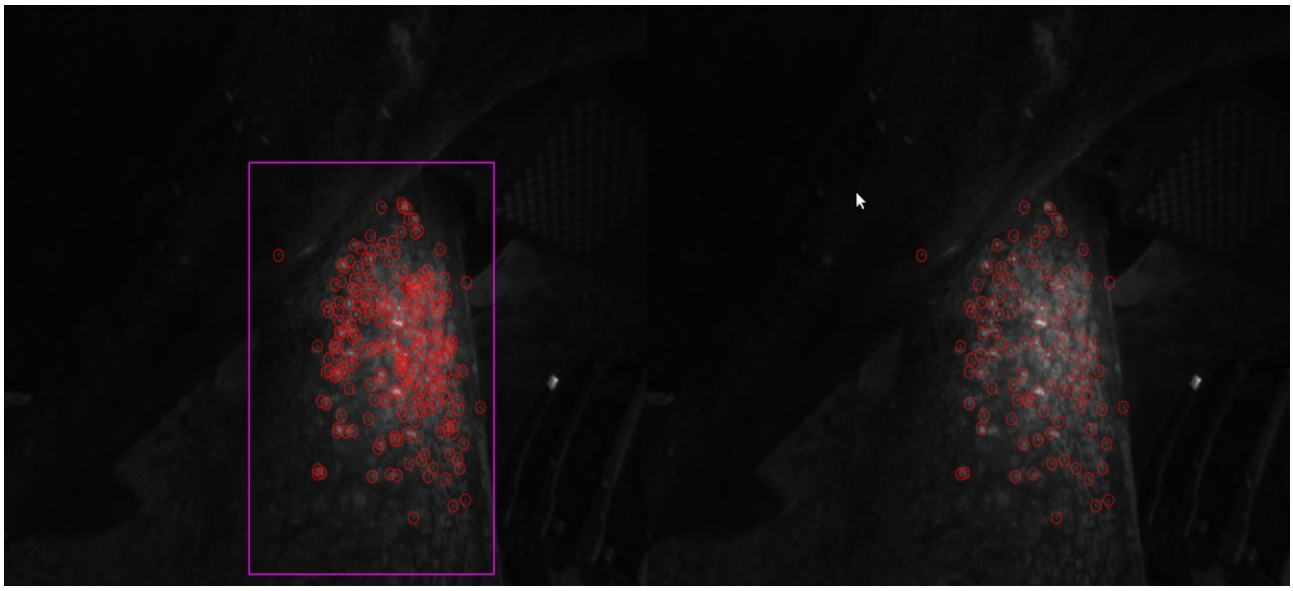


Illustration 15: Adaptive non-maximal suppression applied to thresholded keypoints.

Behrens and Röllinger [23]compared filtering keypoints by strength, ANMS, and a k-d tree based

method. They found that both k-d tree and ANMS significantly reduce the registration error. However since ANMS is computationally more costly, they recommended k-d tree based approach for filtering.

Pseudo code below implements ANMS. It has computational complexity of $O(n^2)$.

```
sort P by strength, let p1 , p2 , ... denote the points in that order
NN.insert(p1)
result = [(∞, p1 )]
//(radius,keypoint)
foreach pi ∈ P do
ri = NN.query(pi )
result.pushback((ri , pi ))
NN.insert(pi)
sort result by radius, return first k entries in result
```

Gauglitz et al [24] presented faster implementation of filtering features spatially called Suppression via Disk Covering. Pseudo code below

```
sort P by strength, let p1 , p2 , ... denote the points in that order
while binary search for suppression radius r do
build NNr on P
result != 0
foreach pi ∈ P do
P = NNr.query(pi)
if pi ∈ P then result = result ∪ pi
NNr.remove(P )
if |result| = k then return result
```

This implementation has runtime complexity of $O(n\log(n))$. Suppression via Disc Covering applied to same frame as before, can be seen in image 16.

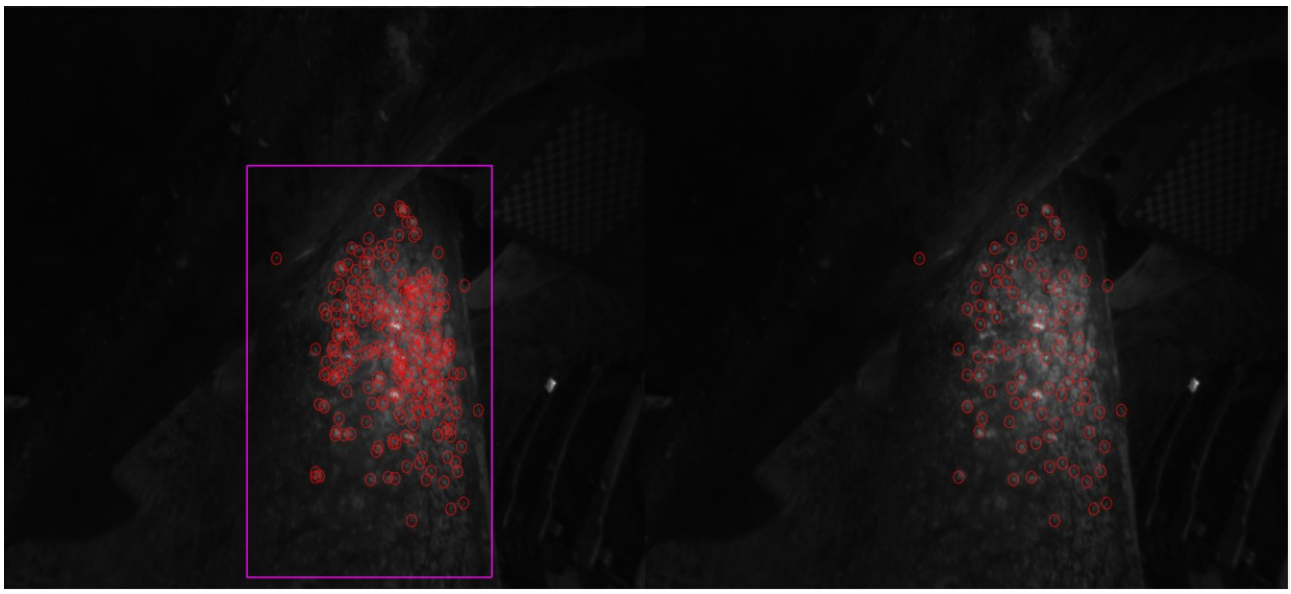


Illustration 16: Suppression via Disc Covering applied to thresholded keypoints.

These two approaches do not produce identical result. At first glance, resulting keypoints may seem same, but looking closer reveal some differences.

2.7 Correspondence problem

Reconstructing structure from stereo requires finding corresponding points in the left and the right image. This can be achieved by block matching or by using feature descriptors. Both techniques require that most interest points are visible in both image and image regions are similar. These both conditions are met in case of stereo rig where baseline is small compared to distance to objects to be viewed.

2.7.1 Block Matching

Block matching uses feature point locations to start with. Template is extracted from a matchee around given point. Matching point is then searched from the image by calculating cost for each possible location. The cost is calculated by traversing all pixel positions in the template and the corresponding points in image.

Most commonly used cost measures are listed in textbook by Cyganek and Siebert[25]. They also shortly analyze weaknesses and strengths of the cost measures. While SAD and SSD are simple to compute, they are sensitive to noise, changes in illumination conditions or camera characteristics. Zero mean normalized sum of squared differences or covariance-variance are better choices when conditions are not ideal. This comes at the cost of more complex computation.

Name of cost function	**Formula**
Sum of Absolute Differences	$E=\sum_{x,y\in W}\lvert T(x,y)-I(x,y)\rvert$
Sum of Squared Differences	$E=\sum_{x,y\in W}(T(x,y)-I(x,y))^2$
Normalized Sum of Squared Differences	$E=\frac{\sum_{x,y\in W}(T(x,y)-I(x,y))^2}{\sqrt{\sum_{x,y\in W}T(x,y)^2\sum_{x,y\in W}I(x,y)^2}}$
Zero Mean Normalized Sum of Squared Differences	$E=\frac{\sum_{x,y\in W}[(T(x,y)-\overline{T}_w)-(I(x,y)-\overline{I}_w)]^2}{\sqrt{\sum_{x,y\in W}(T(x,y)-\overline{T}_w)^2\sum_{x,y\in W}(I(x,y)-\overline{I}_w)^2}}$
Covariance-Variance	$E=\frac{\sum_{x,y\in W}(T(x,y)-\overline{T}_w)\cdot(I(x,y)-\overline{I}_w)}{\sqrt{\sum_{x,y\in W}(T(x,y)-\overline{T}_w)^2\sum_{x,y\in W}(I(x,y)-\overline{I}_w)^2}}$

Table 4: Common cost measures used in template matching.

In table 4 ***I*** stands for intensity values in the image and ***T*** stands for intensity values in the template. Furthermore $\overline{I}_w$ stands for mean of intensity values over the window in the image and $\overline{T}_w$ stands for mean of the intensity values over the window in the template.

In fact zero mean normalized sum of squared differences or covariance-variance are equal measures as is showed below by typing normalized squared error formula open [26]

$$E_{ZNSSD} = \sum_{x,y \in W} \left(\frac{T(x,y) - \overline{T}_w}{\sqrt{\sum_{x,y \in W} (T(x,y) - \overline{T}_w)^2}} - \frac{I(x,y) - \overline{I}_w}{\sqrt{\sum_{x,y \in W} (I(x,y) - \overline{I}_w)^2}} \right)^2$$

$$= \sum_{x,y \in W} \left(\frac{(T(x,y) - \overline{T}_w)^2}{\sum_{x,y \in W} (T(x,y) - \overline{T}_w)^2} - 2 \frac{(T(x,y) - \overline{T}_w) \cdot (I(x,y) - \overline{I}_w)}{\sqrt{\sum_{x,y \in W} (T(x,y) - \overline{T}_w)^2 \sum_{x,y \in W} (I(x,y) - \overline{I}_w)^2}} + \frac{(I(x,y) - \overline{I}_w)^2}{\sum_{x,y \in W} (I(x,y) - \overline{I}_w)^2} \right)$$

$$= 2 - 2 \frac{\sum_{x,y \in W} (T(x,y) - \overline{T}_w) \cdot (I(x,y) - \overline{I}_w)}{\sqrt{\sum_{x,y \in W} (T(x,y) - \overline{T}_w)^2 \sum_{x,y \in W} (I(x,y) - \overline{I}_w)^2}}$$

$$= 2(1 - E_{ZNCC})$$

Table 5 shows example of block (template) matching. Cost of both zero mean normalized sum of squared differences and covariance-variance are shown in last column. Comparing these images gives visual clue that these measures could be same but inverse. Position of best match is marked by red square in image.

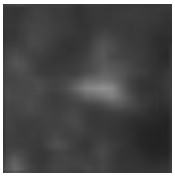
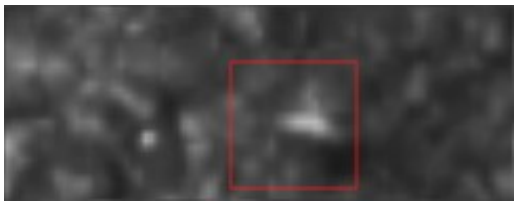
Template	Image	Score
		 

Table 5: Example of template matching. Upper score is normalized cross-correlation. Lower score is sum of normalized error.

The location of the cost function minimum is searched in the final step of block matching. This is easily implemented by keeping track of the score while computing cost function values at image locations.

2.7.2 Descriptor matching

Descriptor matching is nearest neighbor search on extracted descriptors. Some similarity measure is used in searching the best match. Hamming distance can be used for binary descriptors. Euclidean distance can be used in real valued descriptors.

Nearest neighbor search can be solved by exhaustive search. Distance is computed to every other point, keeping track of the best so far. Simple to implement but this approach has high runtime complexity. Approaches designed for better performance are space partitioning methods like kd-tree or approximate nearest neighbor methods like FLANN [27].

2.8 Structure from stereo

Structure from stereo techniques can be divided into two categories: dense and sparse. Dense techniques calculate depth for each pixel of image, creating a depth image. Sparse techniques calculate 3D position for keypoints (interest points) creating a sparse point cloud.

Structure from stereo techniques use extensively concept of disparity *d*. In the case of rectified stereo images disparity is simply the difference between corresponding points along x-axis. In ideal case difference along y-axis should be zero.

$$d = x_l - x_r$$

Intermediate result in creating depth image is the disparity map. Disparity is calculated for each pixel in the disparity map. This gives possibilities for additional constraints. Order of pixel on scan line can be used as additional constraint or possibly abrupt changes in disparity value are not allowed.

Corresponding points are found by similarity measure when creating sparse point cloud. Similarity measure can be area based like block match or it can be based on descriptor calculated for keypoints. In the end 3D positions are calculated from disparities in sparse point cloud too.

Implementation associated with his thesis uses sparse point cloud.

2.8.1 Triangulation

Triangulation is calculating 3D positions based on corresponding image points. Triangulation uses calibration data of the cameras as well. In principle this is straightforward calculation. However noise in locations of the interest points leads to that the projected rays rarely intersect.

In noiseless situation 3D coordinates of the keypoint can be calculated by following equations.

$$\begin{cases} X = (x - c_x)\dfrac{Z}{f} \\ Y = (y - c_y)\dfrac{Z}{f} \\ Z = f\dfrac{T}{d} \end{cases}$$

Above equations are direct applications of similar triangles. These can be implemented as matrix multiplication [28]. Here $\boldsymbol{X}_{3D}$ is 3D vector in homogeneous presentation. Normalization is needed to get coordinate values in Euclidean space.

$$\boldsymbol{X}_{3D} = \boldsymbol{Q}[\,x\ y\ d\ 1\,]^T$$

$$\boldsymbol{Q} = \begin{bmatrix} 1 & 0 & 0 & -cx \\ 0 & 1 & 0 & -cy \\ 0 & 0 & 0 & f \\ 0 & 0 & 1/T & 0 \end{bmatrix}$$

Rays reprojected from left and right image points do not intervene when noise is present. An early approach was to select midpoint on line segment which is perpendicular to both rays [29]. Image 17 is visualization how mid point is located.

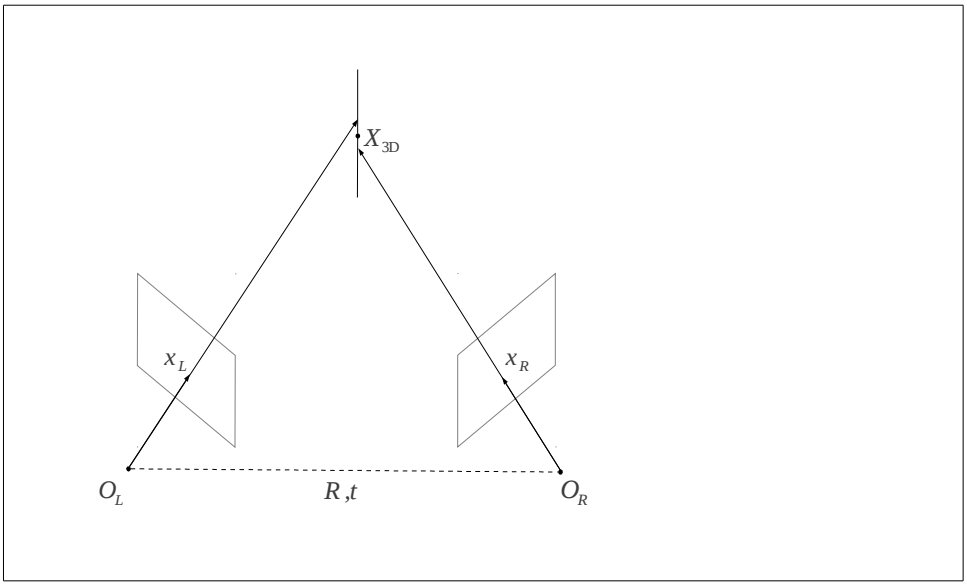


Illustration 17: Example of triangulation by midpoint.

3D point can be solved algebraically too [30]. This method is often called direct linear transform. Remember $\boldsymbol{x}=P\,\boldsymbol{X}_{3D}$. Each keypoint $\boldsymbol{x}=[u\,v\,1]^T$ gives two equations to solve $\boldsymbol{X}_{3D}$ from.

$$u\,\boldsymbol{p}_3^T\,\boldsymbol{X}_{3D}=\boldsymbol{p}_1^T\,\boldsymbol{X}_{3D}$$
$$v\,\boldsymbol{p}_3^T\,\boldsymbol{X}_{3D}=\boldsymbol{p}_2^T\,\boldsymbol{X}_{3D}$$

Moving all to elements to left we get subtractions that equal to zero. Now combining equations from multiple keypoints, 3D point can be solved by finding solution to homogeneous $\boldsymbol{A}\,\boldsymbol{x}=0$ subject to constraint $\|\boldsymbol{x}\|=1$ [31]. Solution is found by singular value decomposition. Solution is singular vector corresponding singular value that equals to zero. When noise is present solution singular vector corresponding smallest singular value.

Triangulation can be also done by finding single 3D point so that reprojection error is minimized, see image 18.

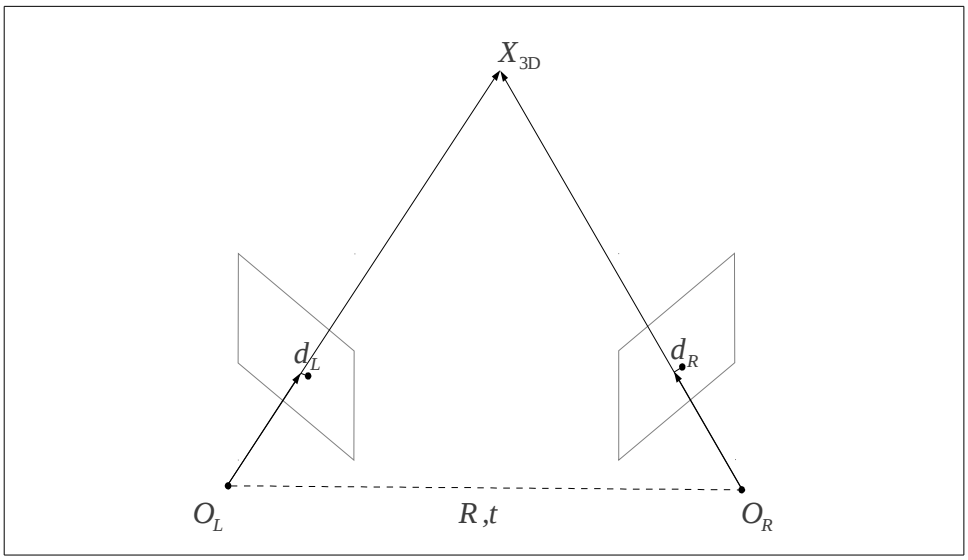


Illustration 18: Example of triangulation by minimizing reprojection error.

Measured image points are noted as $\boldsymbol{x}_L$ and $\boldsymbol{x}_R$. Estimated image points would be $\hat{\boldsymbol{x}}_L$ and $\hat{\boldsymbol{x}}_R$. These points should satisfy the epipolar constraint exactly. Distance between measured and estimated is calculated and solution would now be finding minimum of distances

$$d(x_L, \hat{x}_L)^2 + d(x_R, \hat{x}_R)^2 \; subject \; \text{to} \; constraint$$
$$\hat{\boldsymbol{x}}_L^T \boldsymbol{F} \hat{\boldsymbol{x}}_R = 0$$

Hartley-Sturm [32] devised a method where epipolar plane was parametrized by single parameter. To simplify minimization two rigid transformations are applied. This is possible as rigid transformation does not change distance measure. First transformation moves measured keypoints to image origo. Second transformation places epipoles on the x-axis $(1,0,f_L)^T$ and $(1,0,f_R)^T$. After these changes fundamental matrix takes form

$$\boldsymbol{F} = \begin{bmatrix} f_L f_R d & -f_R c & -f_R d \\ -f_L b & a & b \\ -f_L d & c & d \end{bmatrix}$$

This fundamental matrix fulfills $\boldsymbol{F}(1,0,f_L)^T = \boldsymbol{0}$ and $(1,0,f_R)\boldsymbol{F} = \boldsymbol{0}$. Consider epipolar line passing through point $(0,t,1)^T$. As epipolar line always goes through epipole, epipolar line can be expressed as function of t. Here is used the formula of line defined by two points in homogeneous coordinates.

$$\lambda(t) = (0,t,1)^T \times (1,0,f_L)^T = (tf, 1, -t)^T$$

Details are not presented here but minimum value can be found by elementary calculus. Calculating derivative of distance, derivative turns out to be polynomial of degree 6 in t. Next step is finding roots of this polynomial and evaluating cost function to find global minimum. Durand-Kerner algorithm is used to find roots of polynomial [33], [34].

Kanatani et al found another iterative procedure that forces epipolar constraint [35]. Triangulation by mid-point does not enforce epipolar constraint.

2.9 RANSAC

Many parameter estimation algorithms deal reasonably well with Gaussian noise. However assumption of Gaussian noise is not always true. Non Gaussian noise can lead to poor estimates. For example linear least squares problem is sensitive to measurements that are clearly out of the model. Luckily there exists algorithms that do cope data elements that do not fit assumed model. Random sample consensus (RANSAC) [36] is widely used in computer vision.

Keypoint matching is not flawless operation. Both stereo and temporal matches may include false matches. The use of RANSAC is well justified. False matches are called outliers, correct matches are called inliers.

RANSAC is designed to find most probable parameters from data that contains outliers. RANSAC tries multiple parameter sets and selects the one that best explains given data elements. In meta code RANSAC is described below

```
1. Choose a small subset of data elements
2. Solve model parameters with this subset
3. See how many other data elements fit to the resulting model. Store current
result if largest support
4. Repeat the previous steps till maximum number of iterations reached
```

```
5. Solve parameters using all inliers
```

RANSAC takes in first step random samples of data. Size of subset is minimum needed to solve parameters based on sample. Random selection makes RANSAC non-deterministic. Subsequent runs on same data can give different result. However number of iterations is selected so that this would be improbable.

The number of iterations *k* is often selected empirically but an upper limit can be determined from a theoretical result. Let *p* be probability that the algorithm produces a useful result. Let *w* be the probability of choosing an inlier each time a single point is sampled. Assuming that *s* points are needed for estimating a model and they are selected independently, w^s is the probability that all *s* points are inliers. Then the probability that a sample is contaminated becomes $1-w^s$. That probability to the power of *k* is the probability that the algorithm never selects a set of *s* points which all are inliers and this must be the same as $1-p$. Consequently,

$$1-p=(1-w^s)^k$$

which, after taking the logarithm of both sides, leads to

$$k=\frac{\log(1-p)}{\log(1-w^s)}$$

From this it can be seen that needed number of iterations increases as number of *s* increases. For performance it is beneficial use minimal subset for solving values of parameters.

Value of *w*, that is number of inliers in data per number of data points is rarely well known beforehand. Value can be selected by trial and error or by an educated guess. Implementation in OpenCV uses an approach where value is updated by ratio of support per number of data points.

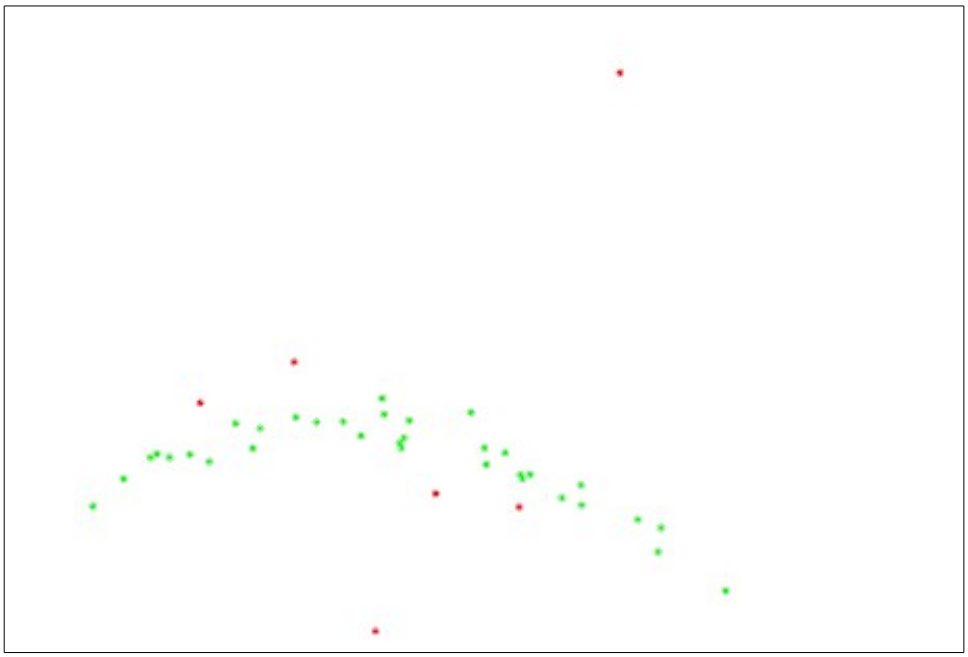


Illustration 19: Inliers supporting ellipses model are plotted in green while outliers are plotted red.

Solving parameters by a given subset depends on the underlying model. Line fitting would require

two samples. Fitting ellipse on data points requires five points. Image 19 illustrates inliers and outliers in finding ellipsis: inliers are pictured in green while outliers are drawn in red. Solving fundamental matrix or essential from stereo images would require 7 or 8 points [37], [14]. After model parameters are solved all data points are evaluated against this model. Each data point is classified either as an inlier or an outlier depending how it fits to model. Number of inliers is called support. RANSAC returns parameters with biggest support. Final step in RANSAC uses all inliers to refine values of the estimated parameters.

Some error threshold is used in the classification to outliers or inliers. Value of threshold affects how RANSAC works. Large threshold increases probability of finding false model. Tight threshold makes estimated parameter value to fluctuate so that adding one element to data may lead to big changes in parameters. [38]

2.10 Absolute orientation

Stereo rig produces 3D point cloud for each time frame by triangulation. Motion between frames can be solved by registration of the point clouds.

Most common 3D point cloud registration algorithms are based on PCA or eigenvectors [39], SVD [40] or iterative closest point (ICP) [41], [42]. PCA based registrations aligns principal directions. SVD based registration gives optimal result if knowledge of correspondence is known. ICP is basically SVD algorithm that updates correspondence of the points on the fly. Work related to this thesis uses SVD based point cloud registration.

First part of the SVD based algorithm is the observation that centroid of point cloud is not affected by rotation. If centroid is subtracted from respective point clouds, only rotation is left to be found.

Rotation is found by using cross-covariance of the point locations. Let $\boldsymbol{P}$ be matrix where i:th column is vector $\boldsymbol{p}_i - \boldsymbol{c}_L$ and $\boldsymbol{Q}$ be matrix where i:th column is vector $\boldsymbol{q}_i - \boldsymbol{c}_R$. Cross-covariance of point locations is $\boldsymbol{M} = \boldsymbol{P}\,\boldsymbol{Q}^T$. Rotation that aligns point clouds is found by finding matrix **R** that minimizes least square distances between points. This same rotation maximizes trace $tr[\boldsymbol{R}\,\boldsymbol{M}]$ [43]. **R** is calculated from SVD of **M**.

$$\boldsymbol{M} = \boldsymbol{P}\,\boldsymbol{Q}^T$$
$$\boldsymbol{M} = \boldsymbol{U}\,\boldsymbol{\Sigma}\,\boldsymbol{V}^T$$
$$\boldsymbol{R} = \boldsymbol{V}\,\boldsymbol{U}^T$$
$$\boldsymbol{t} = \boldsymbol{c}_L - \boldsymbol{R}\,\boldsymbol{c}_R$$

Transformation **R** can be also reflection. This is the case if determinant of **R** is negative. Situation can be corrected by changing sign in third column of **V** and calculating $\boldsymbol{R}$ again.

Finally translation is calculated by subtraction of the centroids transformed into same coordinate system.

2.11 Bundle adjustment

Bundle adjustment refines simultaneously both the structure and camera poses over multiple images. It has been invented in the field of aerial cartography [44], [45]. In aerial cartography scene has small depth changes compared to distance to camera. Bundle adjustment was quickly adopted in close range measurements too, where scene has significant depth changes compared to camera position.

Bundle adjustment is based on minimizing geometric distance between projection of the 3D points and measured points in the image.

$$E = \min_{X,P} \sum_{i \in X} d(\boldsymbol{X_i}, \boldsymbol{x_i})^2 = \min_{X,P} \sum_{i \in X} (\hat{\boldsymbol{x}}_i - \boldsymbol{x_i})^2, \textit{where}$$

$$\hat{\boldsymbol{x}}_i = proj(\boldsymbol{P}\,\boldsymbol{X_i})$$

$$proj: \mathbb{R}^n \rightarrow \mathbb{R}^{n-1}, \; proj(\boldsymbol{x}) = \frac{1}{x_n} \begin{bmatrix} x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Here measured image points are noted as $\boldsymbol{x}_i$ while projected image points would be $\hat{\boldsymbol{x}}_i$. Projected image points calculated in turn from 3D points noted by $\boldsymbol{X}_i$. Camera projection matrices are noted as $\boldsymbol{P}$. Finally *proj*-function realizes normalization of homogeneous coordinates.

Minimum is found by some iterative non-linear solver like Gauss-Newton or Levenberg-Marquardt. These algorithms need partial derivatives of the cost function. Partial derivatives are calculated with respect to keypoint positions and camera poses.

Embed into cost function was normalization. Partial derivative of *proj*-function is

$$\frac{\partial\, proj(\boldsymbol{x})}{\partial\, \boldsymbol{x}} = \frac{1}{x_n} \begin{bmatrix} I_{n \times n} & \begin{matrix} -\frac{x_1}{x_n} \\ \vdots \\ -\frac{x_{n-1}}{x_n} \end{matrix} \end{bmatrix}$$

Next partial derivative with respect to keypoint X is simple. Here *y* is used as intermediate variable that includes rigid transformation of a 3D point.

$$\begin{aligned} \frac{\partial\, d(\boldsymbol{X_i}, \boldsymbol{x_i})}{\partial\, \boldsymbol{X_i}} &= \frac{\partial (\hat{\boldsymbol{x}} - \boldsymbol{x_j})}{\partial\, \boldsymbol{X_i}} \\ &= \left. \frac{\partial\, \hat{\boldsymbol{x}}(y)}{\partial\, y} \right|_{y = R\boldsymbol{X} + t} \cdot \frac{\partial\, R\,\boldsymbol{X} + t}{\partial\, \boldsymbol{X_i}} \\ &= \frac{f}{x_3} \begin{bmatrix} 1 & 0 & \frac{-x_1}{x_3} \\ 0 & 1 & \frac{-x_2}{x_3} \end{bmatrix} R \end{aligned}$$

Intermediate result of partial derivatives with respect to rotation angle is needed for derivation of Jacobian with respect to camera pose. Rotation matrix has nine elements but angle is just one parameter. Calculating partial derivatives of rotation angles around principal axis gives Jacobians below

$$\left. \frac{\partial R_x}{\partial \theta_x} \right|_{\theta_x = 0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -\sin 0 & -\cos 0 \\ 0 & \cos 0 & -\sin 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\left. \frac{\partial R_y}{\partial \theta_y} \right|_{\theta_y = 0} = \begin{bmatrix} -\sin 0 & 0 & \cos 0 \\ 0 & 0 & 0 \\ -\cos 0 & 0 & -\sin 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

$$\left.\frac{\partial R_z}{\partial \theta_z}\right|_{\theta_z=0} = \begin{bmatrix} -\sin 0 & -\cos 0 & 0 \\ \cos 0 & -\sin 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Finally calculating Jacobian of camera pose ϵ. Note that ϵ contains six parameters for camera pose: three parameters for position and three parameters for rotation. Here again *y* is used as intermediate variable that includes rigid transformation of a 3D camera position and orientation. Here is the result

$$\begin{aligned} \frac{\partial d(\boldsymbol{X_i}, \boldsymbol{x_i})}{\partial \epsilon} &= \frac{\partial (\hat{\boldsymbol{x}} - \boldsymbol{x_j})}{\partial \boldsymbol{X_i}} \\ &= \left.\frac{\partial \hat{\boldsymbol{x}}(y)}{\partial y}\right|_{y=R\boldsymbol{X}+t} \cdot \frac{\partial R\boldsymbol{X}+t}{\partial \epsilon} \\ &= \frac{f}{x_3} \begin{bmatrix} 1 & 0 & \frac{-x_1}{x_3} \\ 0 & 1 & \frac{-x_2}{x_3} \end{bmatrix} \begin{bmatrix} I_{3x3} & \frac{\partial R_x}{\partial \theta_x}\boldsymbol{y} & \frac{\partial R_y}{\partial \theta_y}\boldsymbol{y} & \frac{\partial R_z}{\partial \theta_z}\boldsymbol{y} \end{bmatrix} \\ &= \frac{f}{x_3} \begin{bmatrix} 1 & 0 & \frac{-x_1}{x_3} \\ 0 & 1 & \frac{-x_2}{x_3} \end{bmatrix} \begin{bmatrix} I_{3x3} & -[y]_\times \end{bmatrix} \end{aligned}$$

Here combining partial derivatives with respect to rotation forms a skew symmetrix matrix. A more elegant derivation of Jacobians needed in bundle adjustment can be done with the help of Lie algebra also. Such an derivation can be found in Strasdat [46].

2.12 Measuring length

Length of the log equals to total movement under camera. Exact location of the cutting plane is not needed. Position of the cutting plane travels same distance as all the keypoints on surface of log. It is sufficient to sum over movement in each frame to get length of log. That is equal to distance between two cutting planes.

2.13 Visual odometry

Visual odometry is a special case of the structure from the motion problem. Structure from motion refers to a problem where camera trajectories and 3D scene are solved from multiple images. Structure from motion is based on early works like [14], [47]. Alternatively structure from motion may solve distinct camera poses instead of camera trajectories. An example of this is photo tourism [48].

Visual odometry refers to the estimation of the motion using only the input of the camera. Here the 3D structure is omitted. Typical problem setting is to solve the trajectory of a moving vehicle. Early implementations where developed for NASA Mars exploration rovers [49], [50] and [51]. In essence the visual odometry performs same function as wheel odometry. Term visual odometry was coined by Nister in his paper 2004 [52].

Typical visual odometry pipeline is illustrated in image 20. First 2D features are detected after new image is captured. Next correspondence is created between the current and the previous frame. This can be done by feature matching or feature tracking. The feature matching refers to techniques where 2D features are detected independently for each image and matched by some similarity metric. The feature tracking refers to techniques where the appearance of feature is searched in new image using e.g. correlation.

Next phase is the estimation of the motion based on the correspondence. These techniques can be categorized into three groups depending what correspondences are used.

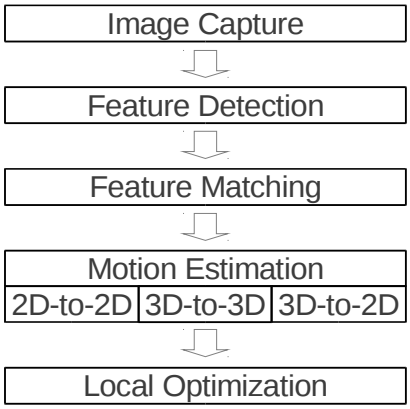


Illustration 20: Typical steps in visual odometry pipeline.

Motion estimation can be performed between two 3D point sets as is done in absolute orientation algorithm. Another possibility is to use correspondence between 3D points in the previous frame and 2D projections in the current frame. Multiple solutions has been found to this problem including direct least squares [53] or by P3P algorithms [54]. Yet another possibility is to use correspondences of the 2D points between the current and the previous frame [55]. This solution is based on calculating essential matrix between the given frames and decomposing essential matrix into rotation and translation.

Iterative refinement can be used as a final step to obtain more accurate estimates.

3 Parallel computing

3.1 Trends in parallel computing

Parallel computing is becoming a common technique. Developers can choose from multiple frameworks when utilizing parallel computing. Current CPUs can natively run multiple threads as 8-10 cores reside on processor. Parallel computing in graphic cards is enabled by publishing interfaces like CUDA or OpenCL. There are also multiple development packages that utilize cloud computing.

Increased performance in CPUs is realized by increasing number of cores on chip. The era of increasing processing speed of CPU - increasing clock frequency - is over. It also means waiting for increased clock speed does not solve performance problems any longer. Learning some parallel programming unlocks door to performance boost.

Operating systems have had threads for ages but using them has been somewhat difficult. Libraries like OpenMP make multi core programming easier. Memory bus becomes easily limiting factor when using OpenMP. Although multiple cores are executing given code there is only one RAM memory on motherboard. CPU caches do help but memory intensive algorithms are affected by speed of memory bus.

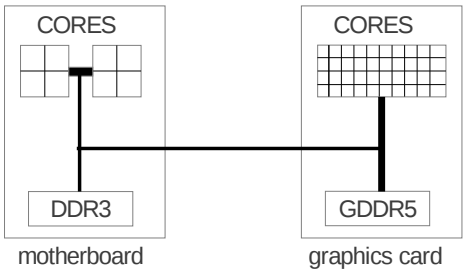


Illustration 21: GPUs run in parallel with CPUs.

Recent addition to parallel computing is acceleration by graphics cards. Parallel computing in GPUs was first designed for 3D graphics drawing. Each pixel had its own GPU thread that computed intensity value. Using CUDA the GPUs can be used for general purpose processing. GPU works as co-processor that run in tandem with CPU, see image 21. NVIDIA made first version of CUDA SDK public in 2007.

CUDA is vendor specific platform and programming model. Khronos Group has developed framework that works on multiple platforms. Name of that framework is Open Computing Language, OpenCL as acronym. First version of OpenCL was published on 2009.

It can be claimed that Moore's law still holds. It has just become harder to benefit of those transistors. In practice parallel computing is not a silver bullet.

This thesis concentrates on graphics card based acceleration. As CUDA has been in use since 2007 it is quite mature framework. Software relating to this thesis is implemented using CUDA.

3.2 Basic principles in CUDA

Concurrent programming in CUDA is implemented by kernels. Kernel is piece of code intended to be run in parallel by multiple CUDA threads. Code sample below shows example of vector addition kernel. This kernel performs simple vector addition C = A + B.

```
// Kernel of vector addition
__global__ void VectorAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

In sequential coding *for* loop would be used to iterate through all vector elements. In CUDA index of vector element is obtained from special variable *threadIdx*, which is provided by CUDA framework. CUDA threads are enumerated starting from zero. So using *threadIdx* gives same effect as looping through indexes. Essentially kernel is executed on all elements of the array. This is called SIMT-architecture: Single-Instruction Multiple-Threads. All the threads execute same instructions but operate on different data. GPU threads do not have branch prediction and have no speculative execution.

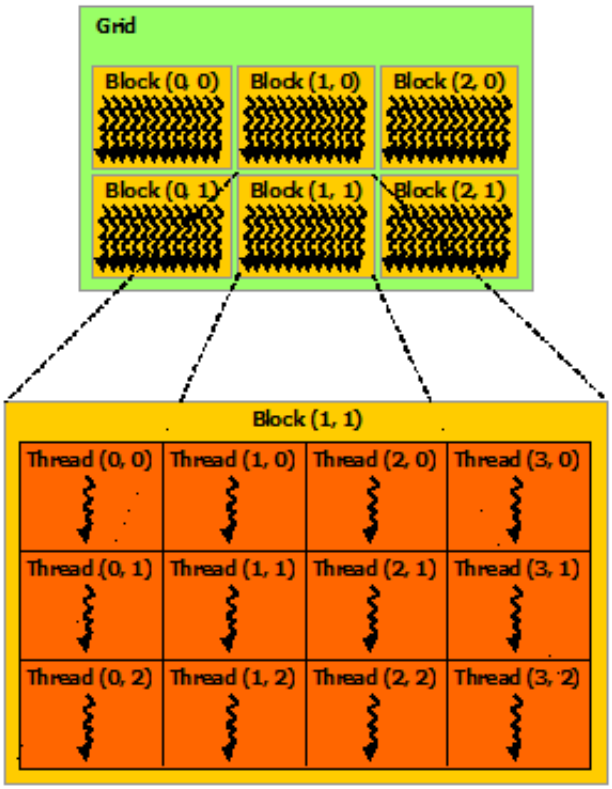


Illustration 22: CUDA runs multiple instances of kernel. In this example grid has 6 blocks and each block contains twelve threads [57]

Threads are grouped into thread blocks and these blocks are executed on streaming multiprocessors on GPU. Group of blocks is called a grid in turn. Image 22 shows kernel instance where each block contains 12 threads and the grid contains 6 blocks. If GPU had 6 or more streaming multiprocessors, all the blocks would be executed simultaneously.

Number of multiprocessors on GPU varies from model to model. This is how scalability is realized by CUDA. It is developers task to divide work (threads) into sufficiently number of blocks. CUDA framework assigns blocks onto multiprocessors. The more multiprocessors are present in the GPU the smaller work queue will be assigned to each one. Actual number of multiprocessors needs not to be known in advance. Assignment of the blocks is handled by framework. It is preferable to give each multiprocessor enough work.

Code sample above did not take into account grid structure of kernel execution. Two additional special variables are needed to get index for vector element, namely *blockDim* and *blockIdx*.

```
// Kernel of vector addition
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}
```

In the sample above index is now calculated from thread id, block id and size of block. Number of CUDA threads does not necessarily coincide with the size of the vector. So thread with id higher than size of vector should do nothing. Hence the if statement in sample.

Thread blocks are assigned to multiprocessor by framework and there is no way of controlling the order of block execution. Also CUDA does not provide simple means for thread blocks to communicate which each other.

To be precise threads are executed in groups of 32 threads, called warps. Threads in a warp are executed simultaneously on the hardware. When CUDA kernel is invoked, the blocks (warps) of the grid are enumerated and distributed to multiprocessors with available capacity. New warps are assigned on the vacated multiprocessors as previous warps terminate. In fact it is preferable to make multiple blocks available to each multiprocessor. Multiprocessors can switch to another block when a memory load or write stops execution.

3.3 CUDA platform

3.3.1 CUDA hardware

The NVIDIA GPU architecture is built around multi-threaded Streaming Multiprocessor. Few generations of streaming multiprocessor architectures has been released since CUDA was first published. Latest architectures are Fermi (SM), Kepler (SMX) and Maxwell. Table 6 lists examples of commercial graphics cards and their computing power.

Fermi SM had only 16 cores while Kepler SMX has 192 cores. So one SMX can host 192 threads simultaneously. Number of other units has been increased as well. In Kepler architecture SMX contains 64 double-precision units, 32 special function units (SFU) and 32 load/store units (LD/ST).

GPU generation	Fermi		Kepler		
Card model (Tesla)	M2075	M2090	K10	K20	K20X
GPU per graphics card	1	1	2	1	1
Multiprocessors	14	16	8	13	14
Cores per multiprocessor	32	32	192	192	192

Total number of cores	448	512	1536(x2)	2496	2688
Multiprocessor type	SM	SM	SMX	SMX	SMX
GFLOPS FP32	1030	1331	2288(x2)	3520	3950
Bandwidths	144	177	160(x2)	208	250

Table 6: Differences between sampled graphics cards based on Fermi and Kepler architectures. [56]

Number of Streaming Multiprocessors in GPU varies depending on the model and price. Table 6 lists number of SM(X) on board too. Bandwidth of device memory is listed as well. High end cards contains more CUDA cores and memory and memory tends to faster compared to cheaper ones.

Image 23 shows principal parts in the Kepler architecture. GPU chip contains a number of multi processors. These all share L2 cache. Each multiprocessor SM(X) contains a number of CUDA cores, in Kepler 192 cores. Each SMX has also L1 cache that doubles as shared memory.

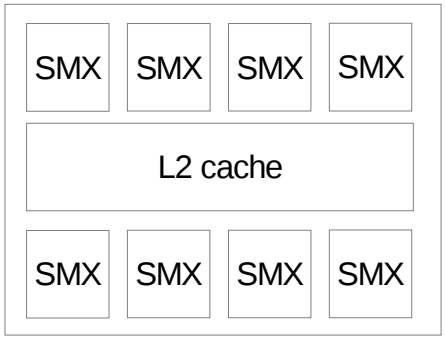


Illustration 23: Schematics of Kepler architecture.

Streaming multiprocessors are designed to execute hundreds of threads concurrently. In the Kepler architecture the number of resident threads per SMX can be higher than 192, but only 192 threads can be served simultaneously, rest of the assigned threads are waiting for their turn.

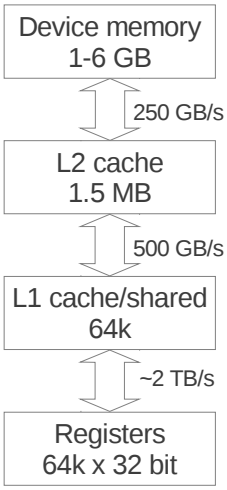


Illustration 24: Memory hierarchy on the GPU.

GPU has multiple types of memory which vary in their bandwidths. Registers are fastest. Second fastest is shared memory or L1 cache. These reside on the SMX chip. L2 cache is shared by all the SMXs on GPU. Finally the bulk of DDR5 memory on GPU is called device memory.

3.3.2 Device memory

Device memory aka global memory is largest bulk of memory on GPU. It is also slowest although it has higher bandwidth than CPU DRAM. Currently DDR5 is used on GPU while DDR3 is common choice on motherboards. Device memory needs to be allocated and values moved from host to device and other way round. CUDA kernels can't access host memory. Host can't access directly GPU memory. CUDA functions has to be used in order to copy values forth and back.

DDR5 global memory has bandwidth of about 200 GB/s. So about 17 GB can be served for simple vector addition of floats per second. GPUs have compute performance around 1T FLOPS. So reading and/or writing to the device memory is bottleneck.

Memory access pattern effects performance of load and store. Access pattern has a direct impact on the performance if kernel is memory intensive. Memory accesses within warp are tried to coalesce into single read or write. 128 bytes would read from device memory if all 32 threads in warp read single-precision float. This would be single instruction if coalescence is possible.

Earlier CUDA releases had very tight rules concerning when coalescing occurs. Image 25 shows different access patterns. In Kepler architecture three access patterns from left can be coalesced into single memory operation. Rightmost access pattern results in two memory operations. Potentially this could reduce performance by 50%. If adjacent threads access memory locations far apart from each other, serialization would occur.

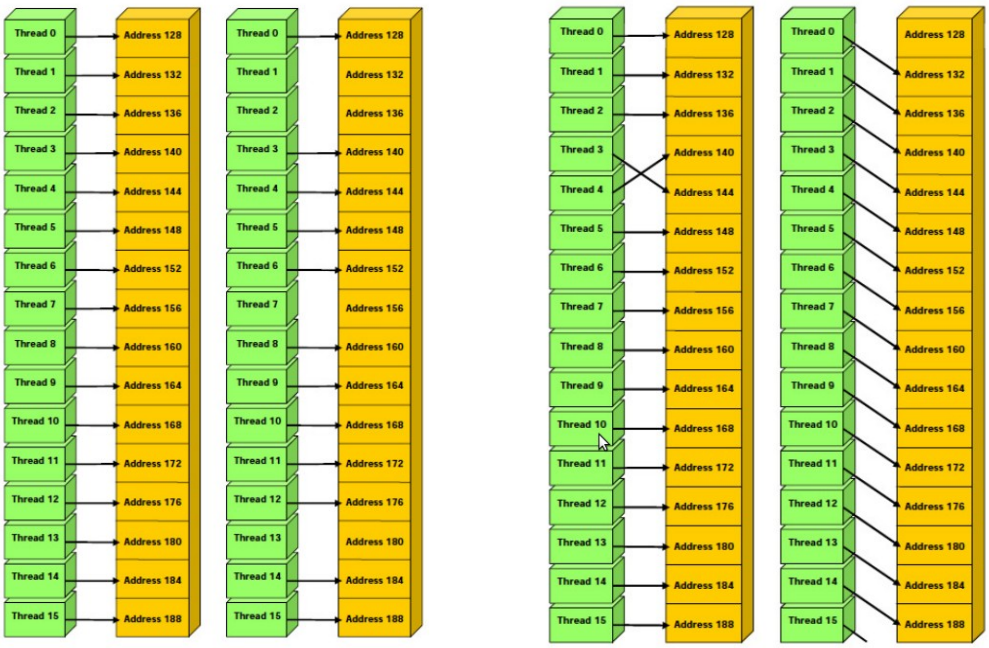


Illustration 25: Examples of memory access patterns from left to right: sequential, divergent, random within warp and misaligned [68].

If coalescing is not possible memory accesses are serialized. In the worst case this could degrade performance by factor of 32 i.e. if there where separate memory operation for each thread.

It is still good practice to use coalesced memory access although L2 cache in Kepler reduces performance hit of non-coalesced memory access.

3.3.3 Shared memory / L1 cache

Shared memory and L1 cache are organized to equally sized memory modules, called banks. Banks can be addressed simultaneously. Memory load or store of n addresses that spans n distinct memory banks can be serviced in one instruction, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank.

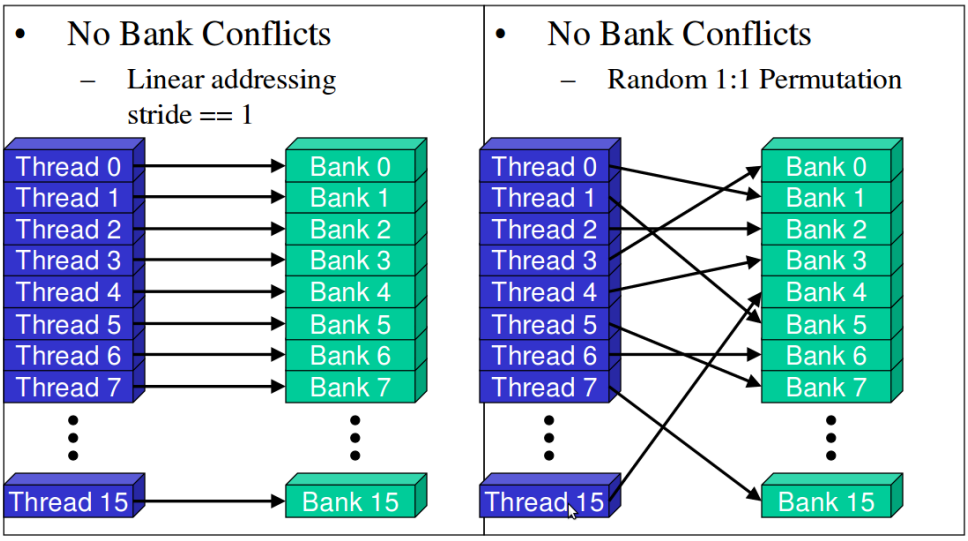


Illustration 26: Examples of shared memory access pattern with no bank conflict [68].

If multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests.

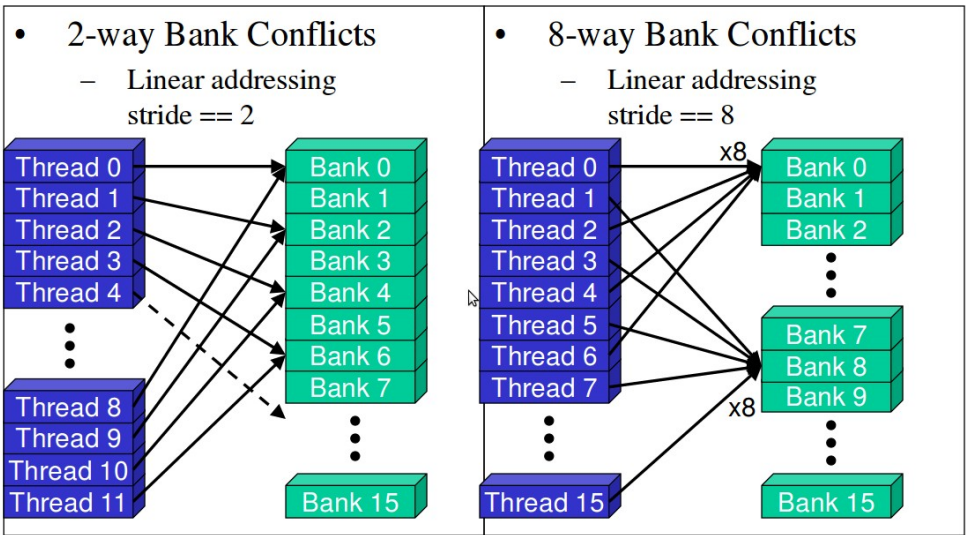


Illustration 27: Examples of shared memory access pattern with bank conflicts [68].

Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per clock cycle.

3.3.4 Registers

Kepler architecture support 64K of 32-bit register values, for example single-precision floating points. CUDA does not have context switch. Instead state of each thread on SMX is kept on register whole lifetime of the block. This enables fast switching from a block to another block. But this comes at a cost. Number of registry entries limits how many threads can be kept active on SMX.

3.3.5 Texture and constant memory

SMX has caches for special memory types like texture and read-only memory.

Using texture cache lifts requirement for memory access patters. Additionally texture cache provides hardware interpolation on float coordinates. Texture also deals with out of bounds addressing, clamping or wrapping.

Constant memory provides fast access when all the threads load the same memory address. In this case no serialization occurs.

3.3.6 CUDA Execution model

CUDA kernels are executed in warps. Warp is group of threads that is executed simultaneously by multiprocessor. Size of warp has been 32 since beginning of CUDA. However it could change in future.

Threads are assigned to multi processors in warps. Warps can be either from same thread block or from different thread blocks. Resources on multi processor are allocated to resident threads/warps/blocks. Resources are limited so there will be upper bounds for their usage.

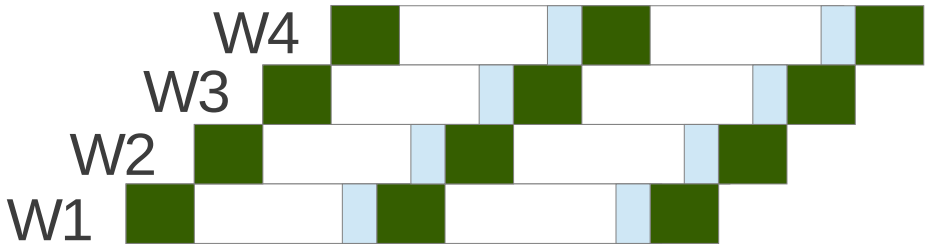


Illustration 28: Example of warp scheduling: green mean executing, white waiting for data, light blue means ready to execute.

Streaming multiprocessor can switch between warps with no apparent delay. Image 28 shows an example with 4 warps. Warp is executed (green) until it is forced to wait for memory access (white) or completes. SMX switches execution to another thread that is ready to execute (light blue). Warp scheduler selects from warps that are in ready to execute state. In example there is no selection as only one warp is ready to execute in turn.

3.4 CUDA compute capabilities

CUDA capability describes properties of the hardware. These properties tell limitations that given CUDA enabled device exerts on kernels. Or the other way round – CUDA capabilities tell what is possible on the given device. Table 7 lists major limitations by capability version.

Both grid and thread blocks are limited by size. Thread block can be contain up to maximum number of threads. Kernel grid can contain up to maximum number of thread blocks. Size of thread-block is limited up to 1024 threads which is relatively low value. However upper limit of grid size is $2^{31}-1$.

For convenience both thread block and grid sizes can be given as a two or three dimensional structure. This simplifies assignment of CUDA threads to data elements when data is two or three dimensional.

Technical specifications	**Compute capability (version)**								
	1.0	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**	**5.2**
Maximum dimensionality of grid of thread blocks	2				3				
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1			
Maximum y-, or z-dimension of a grid of thread blocks	65535								
Maximum dimensionality of thread block	3								
Maximum x- or y-dimension of a block	512				1024				
Maximum z-dimension of a block	64								
Maximum number of threads per block	512				1024				
Warp size	32								
Maximum number of resident blocks per multiprocessor	8					16		32	
Maximum number of resident warps per multiprocessor	24		32		48	64			
Maximum number of resident threads per multiprocessor	768		1024		1536	2048			
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			
Maximum number of 32-bit registers per thread	128				63		255		
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB	96 KB
Number of shared memory banks	16				32				

Table 7: Excerpt of CUDA compute capabilities [57].

Grid size can be very large but not all thread blocks can be assigned to multiprocessors simultaneously. Current GPU could host up to 480 resident thread blocks if GPU contains 15 multi processors (15 times 32). Grid size can be way bigger.

Other properties do not limit parallelism of CUDA code so obviously. These are maximum number of resident blocks per multiprocessor, maximum number of resident warps per multiprocessor and maximum number of resident threads per multiprocessor. Here the keyword is resident. These indirectly limit how many thread blocks can be assigned to multiprocessor.

Number of resident threads per multi processor is limited up to 2048 in GPU with compute capability of 3.0 or later. While maximum number of thread blocks is 32 per multi processor, each block could have up to 64 threads. If more threads per block were used it starts to limit number of resident blocks. Only 2 thread blocks of 1024 threads could be assigned to multiprocessor. Which could severely limit parallelization of kernel.

Number of resident warps per multi processor is limited up to 32 in GPUs with compute capability 3.0 and 3.5. Thread blocks with 32 threads would be limited by this value. Maximum number of threads would give 64 thread blocks. In CUDA capability 5.0 or later maximum number of resident warps per multi processor is 64. This is consistent with maximum number of threads.

Finally the number of registers in multiprocessor is limited to 64K. These registers are allocated to resident threads, possibly in different thread blocks. Maximum number of resident threads was 2048, this gives 32 registers per thread. Number of resident warps is decreased if kernel requires more than 32 registers per thread.

Amount of shared memory affects number of resident warps as well if shared memory is used by the kernel.

3.5 Programming model

CUDA threads are executed on a physically separate device that operates as a co-processor to the CPU. CUDA application contains code that is executed by CPU (host) and GPU device. Code called by host is called host code. Code executed by GPU is called kernel code. Kernel is invoked by CPU.

As a programming language CUDA is extension of C/C++. CUDA is low level programming language like C/C++ is. CUDA consists of a minimal set of extensions to the C language and a runtime library. The CPU host initiates parallel processing of multi-threads by calling kernel functions which are run on GPU.

An example of a kernel was shown previously. Example below demonstrates how the kernel is called. First number tells the size of the grid i.e. how many blocks are used. Second number tells how many threads are in each block.

```
// Kernel call of vector addition
VecAdd<<<1024, 64>>>(A, B, C, n);
```

Kernel code reflects hardware architecture. Especially this can be seen in memory types. Local variables are stored in register file. This applies also to local arrays if their size is known at compilation time. Otherwise local array is stored and loaded from global memory. Different memory types will be used by using respective keywords like *__register__*, *__shared__* or *__constant__*.

Threads in a block can be synchronized by *__syncthreads()*. In general threads in different blocks can't be synchronized. Function call *cudaDeviceSynchrize()* does the trick but that needs to be called by the host.

Currently CUDA comes with few atomic operations. These include addition, subtraction, selecting minimum and maximum. Atomic operations work on the device or shared memory.

Threads in a warp execute synchronously if no warp divergence occurs. This can be utilized by developers.

Threads in the same warp can read each others registers using the shuffle instruction. Reading registers is faster than using shared memory. Careful design is needed: threads can only read data from another thread which is executing same shuffle instruction. Retrieved value is undefined if target thread is inactive for example due warp diverge. Warp shuffle was introduced in Kepler architecture.

3.6 Best practices

This chapter describes design principles that have been found recommendable by CUDA community. These are not mathematical or theoretical results but rather conventions that many authors have found preferable.

3.6.1 Minimizing data transfer overhead

It is preferable to minimize data transfer between the host and the device memory. This is drawback of using separate fast memory memory on the graphics card. Data needs to transferred to and from memory of the graphics card.

As corollary, intermediate data structures should be allocated, operated and destroyed on the device memory. Also batching many small data transfers into one bigger data transfer gives a performance boost as each transfer contains some overhead.

Additionally there are few techniques available to speed up memory transfers. Namely asynchronous data transfer and page locked memory.

Data transfers between the host and the device are blocking. Control is returned to the host only when transfer is complete. CUDA framework provides a non-blocking transfer too which returns control to the host immediately. So the host can invoke kernel before transfer is completed. Data and operations on it can be interleaved. Asynchronous data transfer uses *streams* of CUDA framework.

CPUs implement virtual memory address space so that software can address more memory than is installed on motherboard. 32-bit operating systems can address up to 4GB RAM while 64-bit operating system can address up to 16 exibytes of RAM. Modern computers can easily provide 4GB of RAM but $(4GB)^2$ is too much to be physically installed. Memory pages not fitting on physical RAM are swapped on hard drive.

Pinned aka page locked memory resides always on physical RAM. Using pinned memory speeds up GPU data transfers as an extra step can be excluded. However pinned memory is limited resource. It should be used with consideration.

3.6.2 Latency hiding

Memory reads and writes take multiple clock cycles. CUDA thread executes until waiting memory operation to finish is the only option. Memory latency can be counteracted by maximizing number of concurrent threads in multiprocessor (thread level parallelism; TLP) or by giving each thread as many instructions to execute as possible (instruction level parallelism; ILP).

Threads are assigned to multiprocessor in warps that is a group of 32 threads. All assigned warps are called resident warps. Only one warp is executed at time. When all threads in warp are stalled warp scheduler switches to another warp. So assigning as many warps as possible in multiprocessor helps hiding memory latency.

As a guideline for thread level parallelism, value of occupancy can be calculated. Occupancy is the ratio of resident warps to maximum number of resident warps. The denominator is fixed by CUDA capability that graphics card supports. The numerator and thus occupancy is determined by the resource limits in the streaming multiprocessor

- Maximum number of blocks per SM(X)
- Maximum number of threads per SM(X)

- Maximum number of warps per SM(X)
- Amount of shared memory
- Amount of registers

However higher occupancy does not always mean higher performance. Low occupancy always means poor hiding of memory latency.

Another way of hiding memory latency is to give instructions for threads to execute. Thread is not stalled when memory read or write is reached. Thread is stalled when required memory value is needed and not yet available. If kernel code contains instructions that do not depend on read or write they will be executed.

```
x = a + b; // memory read issued
y = a + c; // independent, can start anytime
z = x + d; // dependent, must wait for completion of x (stall)
```

Volkov has used ILP among other techniques in his works [58], [59]. In BLAS routines he achieved higher performance than original library by NVIDIA. Later his work was adopted by NVIDIA's BLAS library.

3.6.3 Maximizing memory bandwidth

Performance can be boosted by using as fast memory as possible. Registers being fastest, next shared memory and last device memory. Each memory type has it own limitations which need to considered when using them.

Registers are a scarce resource. Although being fastest memory available extensive usage can degrade performance: registers are assigned to all the warps at the multiprocessor. Register usage can become limiting factor in the occupancy. The compiler can be advised to use given number of register in the kernel code. This occurs at the expense of instructions. The maximum number of registers per thread can be limited per-file using the *-maxrregcount* option or per-kernel using the *__launch_bounds__* qualifier.

If all local variables do not fit into the register file, values are stored and read from the device memory. As a side-effect performance is significantly reduced. Developer should be careful about the number of local variables. Register spilling into device memory should be avoided.

Bank conflicts should be avoided when using shared memory. Coalescing should be pursued when reading from or writing to the device memory. In other words, strided memory access should be avoided.

Textures are beneficial when reading between integer indexes as it provides hardware accelerated interpolation of a pixel value.

Straightforward rules do not exist on how to use the CUDA memory hierarchy. Experimentation is needed to find the best design.

3.6.4 Instructions and control flow

Control flow statements (*if, switch, for, while, do*) potentially degrade performance. This happens when threads in warp take different execution paths. Otherwise threads in warp execute synchronously.

Instructions in different execution paths are serialized when threads in warp take different execution

paths. Other threads will be waiting their turn. When all different execution paths have completed, the warp converges again to one execution path. However the number of instructions executed by warp is increased by warp divergence.

Performance degradation related to control statements can be mitigated if the controlling condition is multiple of the warp size. In other words all the threads in the same warp should take same execution path. Threads in next warp could take another path, but same wihtin warp again. Obviously this mitigation can be rarely used. For example no warp divergence occurs if condition depends on thread-idx per warp size.

The compiler may also use branch predication. In this case no divergence occurs. Code sample below demonstrates how branch predication works. Given if-statement below

```
if (x < 0.0)
      z = 0;
else
      z = sqrt(x);
```

will be compiled into following predicated sequence of instructions.

```
      p = (x < 0.0)
p:    z = 0;
!p:   z = sqrt(x);
```

All threads in warp execute three lines above. Execution cost is sum of both branches as all threads execute both branches. The compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by the branch condition is less than or equal to a certain threshold: If the compiler determines that the condition is likely to produce many divergent warps, this threshold is 7; otherwise it is 4 [60].

Mathematical functions like sin, cos, pow and exp come in two flavors. Function calls prefixed with double underscores, *__functionname()*, use hardware level implementation. They are faster but less accurate than implementation called by f*unctionname()*. Hardware level math functions can be force by use *-use_fast_math* compiler flag.

There are other considerations which are not covered here. For details, best practices guide by NVIDIA should be studied [60].

3.6.5 Heuristic conventions

As can be seen there are no straight forward rules when designing CUDA enabled applications. Some experimentation is needed for good outcome. Some rules of thumb can be given about both kernel design and launch configuration. These both affect performance.

Rough guidelines regrading kernel design can be found in CUDA literature. Here are three golden rules using CUDA by Rob Farber [61]

- transfer data to GPU and keep it there
- give GPU enough work to do
- reuse objects already in GPU

Heuristic rules regarding kernel launch configuration can also be applied. These concern how many blocks should be used in a grid and how many threads should be used in a block.

The number of blocks should be at least greater than number of multiprocessors. All the multiprocessors would have at least one block to execute. Preferably number of blocks per

multiprocessor should be at least two: multiple blocks can run concurrently in a multiprocessor. Number of blocks should be at least 100, if designing for future. This would guarantee then kernel would scale in future devices.

Number of threads per block should be multiple of warp size. This avoids wasting computation on under-populated warps. A minimum of 64 threads per block should be used, and only if there are multiple concurrent blocks per multiprocessor. Between 128 and 256 threads per block is a better choice and a good initial range for experimentation with different block sizes. Use several (3 to 4) smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call *__syncthreads()*. This all depends on your computation, so experiment!

Note that when a thread block allocates more registers than are available on a multiprocessor, the kernel launch fails. Kernel launch will fail if too much shared memory or too many threads are requested.

3.7 Parallel Algorithms

This section introduces parallel algorithms used and studied in the scope of this thesis. In general this would all too large theme to be addressed in a thesis work.

Some problems can easily be separated into number of subproblems. These subproblems can be run in parallel. An example of this would be stereo rectification: each rectified pixel can be computed regardless of other pixels. In parallel computing this is called embarrassingly parallel problem. This is often the case where there exists no dependency between those subproblems.

3.7.1 Reduction

Reduction refers to computing single value from all the elements of array. Examples of such computation are finding minimum or maximum in array or calculating sum of elements. These kind of general calculations are building blocks for more complex algorithms.

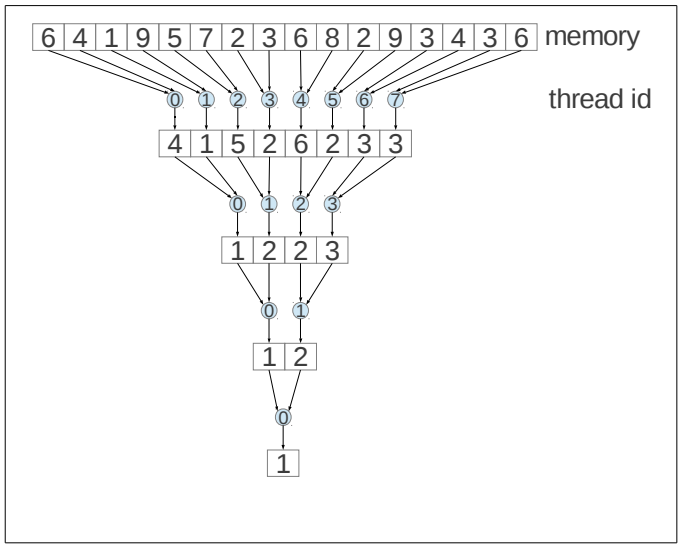


Illustration 29: Finding minimum by using multiple threads.

In sequential computation reduction would be simply implemented by looping over all the elements in array. How to implement parallel reduction? Reduction involves comparing/processing elements with each other. So computations are not independent of each other, which makes utilizing multiple threads more demanding.

Parallel reduction uses CUDA threads in binary tree fashion, see image 29. Each thread takes two elements of array and performs reduction for them and are stores the result back to memory. In next step only half of the threads are needed to perform remaining reduction. And so forth until single value is returned. This results in log(N) steps where N is number of elements.

Naive parallel implementation does not take into account hardware limitations of CUDA architecture. Mark Harris has elaborate explanation on implementing parallel reduction on CUDA and how to optimize its performance [62]. First of all it is beneficial to address elements so that adjacent elements are addresses by adjacent threads, see image 30. This way coalesced memory addressing is achieved and bank conflicts of shared memory are avoided as well.

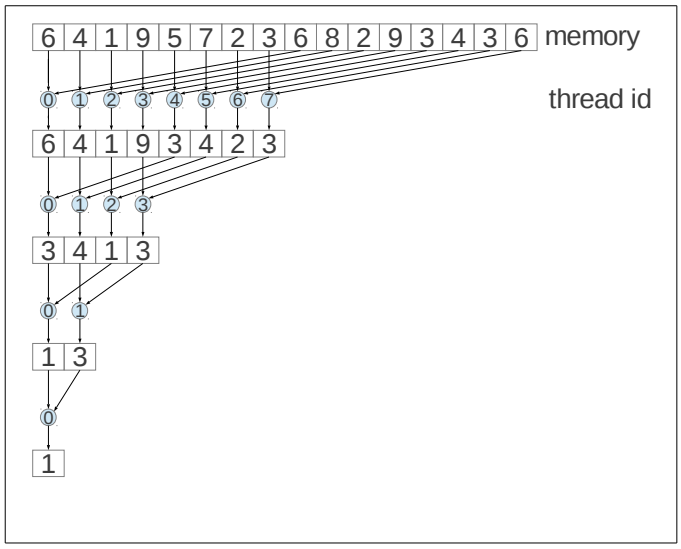


Illustration 30: Reduction by using optimized memory addressing.

Luitjens [63] developed implementation that is based on warp shuffles. This is beneficial for small arrays or in the final steps reducing large array.

Number of threads has upper limit of 1024 per block. Larger arrays require using multiple blocks and solve communication between blocks. CUDA does not provide mechanism so that threads in different blocks could communicate with each other. There are multiple solutions: larger arrays can be processed by using two phase reduction. Second reduction operates on results of first reduction [64]. Or results of blocks can be combined by using atomic operations.

Image 31 shows performance of different reduction (sum) implementations. For comparison single threaded CPU reduction is plotted too. It appears that parallel reduction performs faster only on an array of 100 thousand elements and more. Below that reduction performance is almost constant and about two decades slower than sequential reduction. In addition there appears to be no significant difference in techniques used to implement reduction.

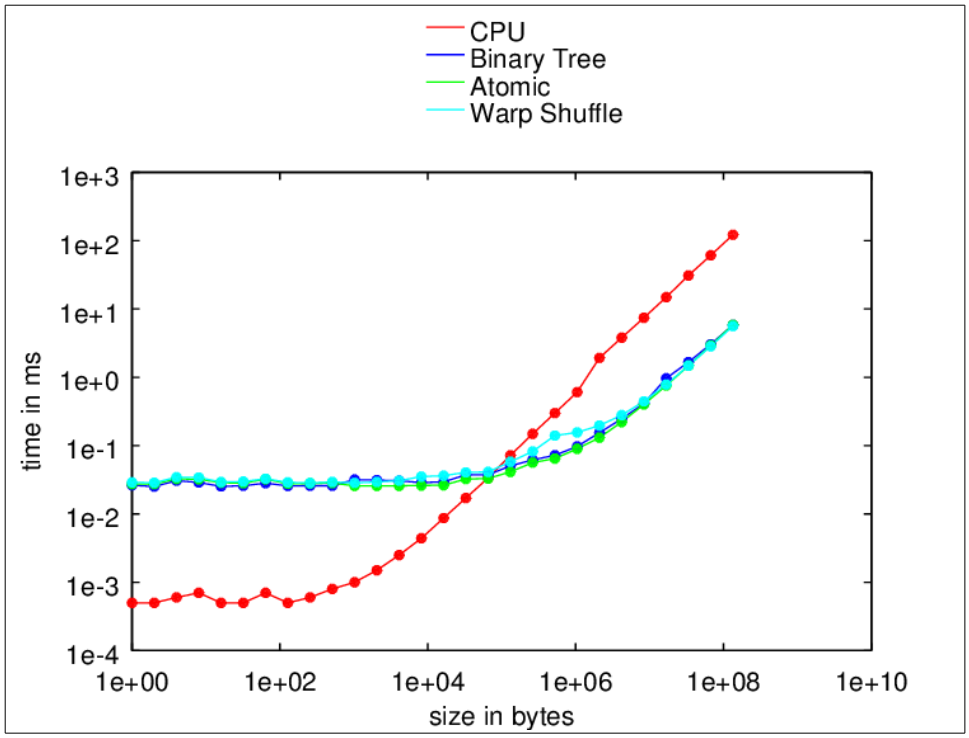


Illustration 31: Performance of parallel reduction algorithms compared to sequential reduction.

3.7.2 Bitonic sort

Sorting is frequently needed algorithm in software engineering. In sequential world quick sort is often selected choice of sorting algorithms. In parallel world there are better options available, like bitonic sort [65]. Bitonic merge sort became better known after addressed in GPU Gems [66].

Bitonic sort is based on properties of bitonic sequence. Bitonic sequence is monotonically increasing or decreasing between min and max elements, see examples in image 32. Indexes of elements roll over from end to begin when considering monotonicity.

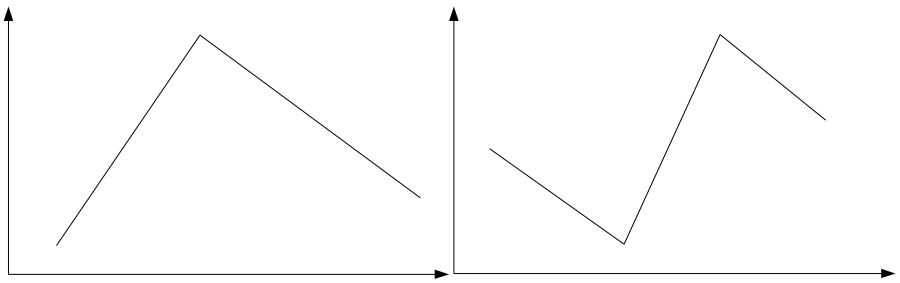


Illustration 32: Examples of bitonic sequences.

Bitonic sequences can be sorted by using recursively bitonic split. In bitonic split a sequence is split in two parts and pairwise min-max comparison is executed on elements in subsequences. Elements in subsequences are swapped if needed. Resulting subsequence has property that all elements are smaller (or greater) than elements in other subsequence.

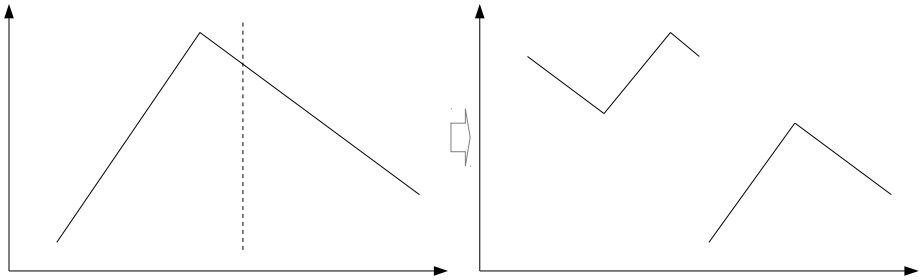


Illustration 33: Applying bitonic split on bitonic sequence result in two bitonic sequences.

Resulting subsequences are bitonic. Now sorting bitonic sequence is straight forward. Sequence is sorted by using bitonic split recursively to smaller and smaller subsequences until finally splitting subsequence of two elements. Bitonic sequence can be sorted in log(n) steps of recursions where n is number of elements.

Generally sequences are unsorted. Unordered sequence should be converted to bitonic sequence before bitonic split can be used. Transforming is performed by bitonic merge. Bitonic merge takes two bitonic sequences and sorts them either to increasing or decreasing order by using bitonic split. Sorting can be increasing or decreasing. Merging is started by sorting two element arrays – which is trivial. Sorting order is alternated to achive bitonic sequnce. In second stage two element arrays are concatenated to 4 element bitonic arrays, which can sorted again by bitonic split. Sorting order is alternated as in previous step. Sequence of n elements can be sorted by using bitonic merge from bottom to to top until whole sequence is ordered.

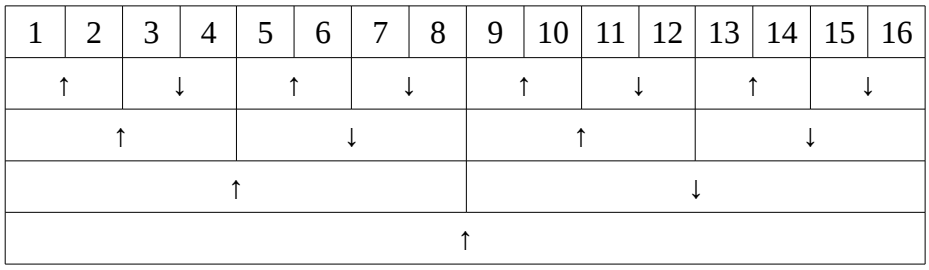


Illustration 34: Sorting 16 elements sequence by bitonic merge.

Bitonic sort works naturally for sequences with length of power of two. Sorting arbitrary length sequences is performed by padding up to next power of two.

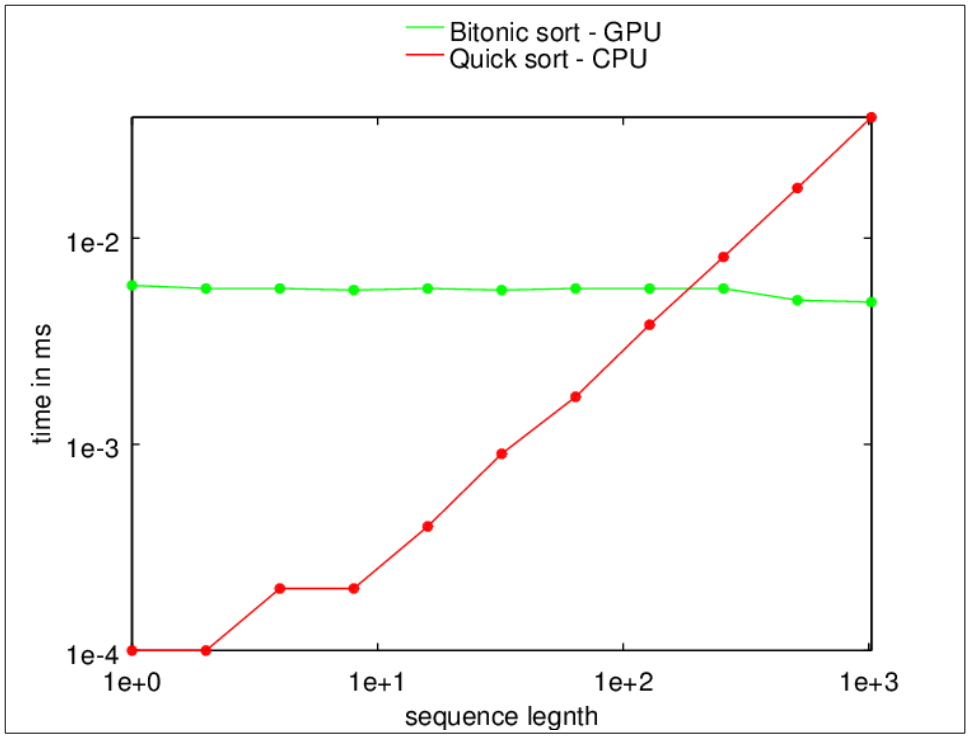


Illustration 35: Performance of bitonic sort compared to quick sort.

Bitonic sort is easy to parallelize. Pairwise comparison and swap is assigned to CUDA thread. Plot 35 compares performance of single threaded quick sort to parallelized bitonic sort.

Bitonic sort gives virtually constant response time while response of quick sort increases when number of elements increases. Length of sequence is limited to arrays that can be sorted by single thread-block. Maximum number of elements is 1024.

4 Log measuring

This part of the thesis describes hardware and software used for log length estimation. First image capturing hardware is described. Next topic is nature of the images captured. After that computing platform and used software packages are presented. Next chapter describes developed measuring software. Last chapter comments accuracy of the measurements.

4.1 Capturing hardware

Capturing images has been part of earlier thesis work. This chapter shortly describes hardware and procedure of the image capture.

Images of log were taken by high speed camera and synchronized flash to prevent motion blur. Frame rate of camera was 120 Hz. This high frame rate was needed as log is moving several meters per second. Synchronous flash is able to overpower ambient lighting and make stop-motion effect apparent without the need for dark ambient operating conditions.

Image 36 shows built stereo rig. Cameras can be seen at left and right side. Flash is placed in between them. Whole equipment was placed inside sturdy metallic casing.

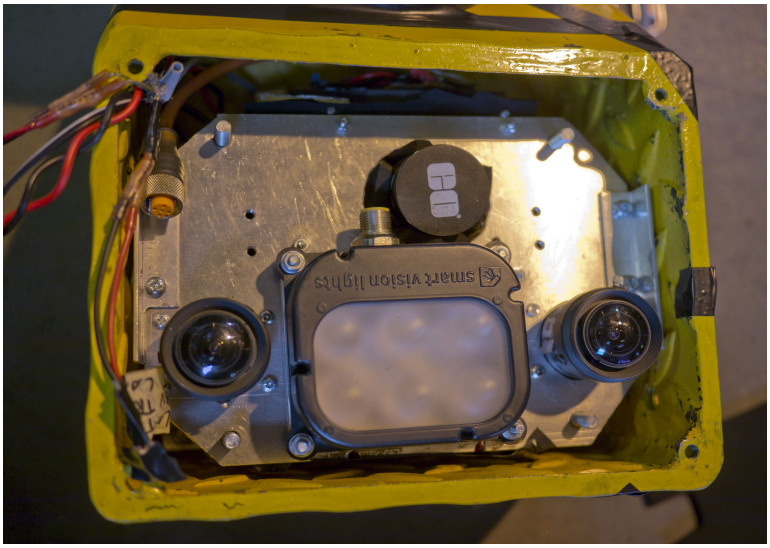


Illustration 36: Stereo image capturing device.

Image 37 shows how measurement device is mounted on harvester head. Images were captured from above diagonally. Cameras and flash light are protected by optical plexiglass. Protection is needed as fragments of bark and wood could splint when log is moved by feeder.

Illustration 37: Mounting of stereo rig on harvester head.

Plexiclass is not supposed to effect rays of capture. However this is one possible source of error. Stereo calibration was performed without plexiclass while measurement images were taken pexiglass in place.

4.2 Stereo images

Modern harvesters are capable to feed log at rate of 6 m/s. This means movement of 50 mm per frame when log is moving at full speed. However using maximum speed at feeder reduces the quality of the delimbing. In practice feeder speeds of 4-5 m/s are used.

Illustration 38: Three subsequent frames taken from harvester head.

Image 38 shows series of three subsequent frames. Interest points move several pixels from frame to frame. In fact movement can be tens of pixels. This means that visual flow based methods can't be used in this problem.

4.3 Processing platform

Images were processed by the following hardware

- AMD Athlon x4 2600 GHz

- GPU GeForce GTX 660

Used GPU supports CUDA capability version 3.0 and comes 5 streaming multi-processors. Altogether number of available CUDA cores is 960. Amount of device memory is 2GB. This hardware comes with moderate performance capability. These are not high-end devices.

Following software packages were used in image processing

- Ubuntu 12.04 LTS
- CUDA 5.0 and 6.0
- OpenCV 2.4.5
- Qt 4.8.1

Development tools included

- Qt Creator 2.4.1
- NVIDIA Visual profiler 5.0 and 6.0

Ubuntu is a desktop operating system. Real-time operating system would be needed if visual measurement would be used for feedback, controlling movement of the log. It was not covered in this thesis.

4.4 Measurement pipeline

Two implementations of log measurement software were implemented: single threaded and GPU accelerated. Single threaded implementation serves as a baseline and verification for GPU accelerated implementation.

UML activity chart 39 depicts pipeline for stereo frame processing. Essentially keypoints are matched both spatially and temporally. This results in two point clouds and log movement is estimated by aligning these two point clouds. Finally log length is obtained by summing over movements per frame.

Distortion removal and stereo rectification is performed as preprocessing step. Removing distortion and rectification can be done with the same remap function. This approach is used in OpenCV [67]. Some smoothing is also included as bilinear interpolation is used to estimate non-integer pixel values.

Next step finds possible keypoints in the current stereo frame. Here FAST is used for its speed and invariance for illumination changes.

Keypoint finding is followed by finding stereo matches which is realized by the block match. Cost function of block match is normalized cross correlation. Window where template is searched is narrowed by epipolar line. This is further narrowed down by estimating the possible location by using plane induced homography. Threshold of cost value is used to rule out impossible matches.

3D points can be reconstructed after correspondence in stereo are found. Stereo matches are corrected by the approach explained in 2.8. This forces the epipolar constraint. After this 3D positions can be simply triangulated as they fulfill the epipolar geometry perfectly. A point cloud is achieved as a result.

BRIEF is used as binary descriptor which is used in temporal match. BRIEF is simple to calculate which is good for performance. BRIEF is also insensitive to illumination changes. BRIEF is

sensitive to rotations but in this particular application rotations are minimal.

Next temporal matches are calculated. Descriptors in current frame are matched against descriptors in previous frame.

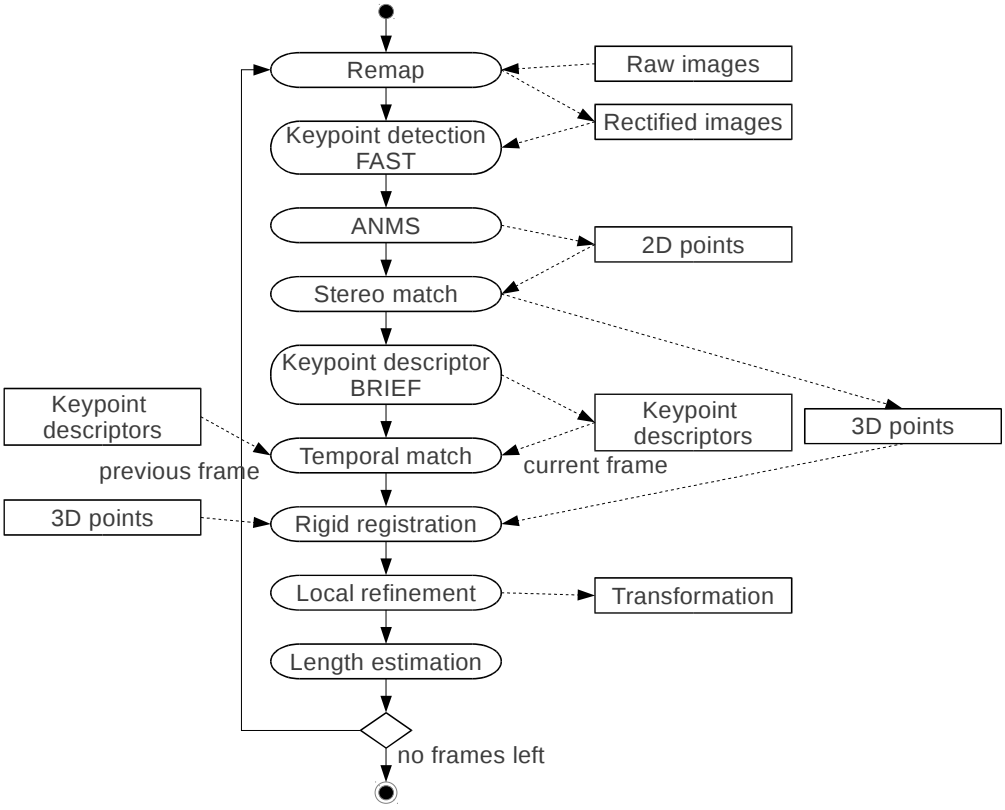


Illustration 39: UML activity chart for single threaded measurement.

Correspondence between 3D points in two subsequent frames has been initiated after stereo and temporal matches are found. To find movement between frames algorithm for absolute position can be used to along with RANSAC. RANSAC is needed as both stereo and temporal matches can include outliers.

Movement between frames is refined before length estimation. Refinement step uses techniques related to bundle adjustment. In this implementation re-projection errors are minimized only in two frames: current and previous.

UML diagram in picture 40 depicts GPU accelerated pipeline used to measure log movement under stereo camera. This is essentially the same as single threaded. Most steps use GPU kernel to speed up execution. GPU kernels are used in a synchronous fashion. That means that a kernel call blocks the execution of CPU until the kernel completes. So the diagram is quite similar to single threaded implementation.

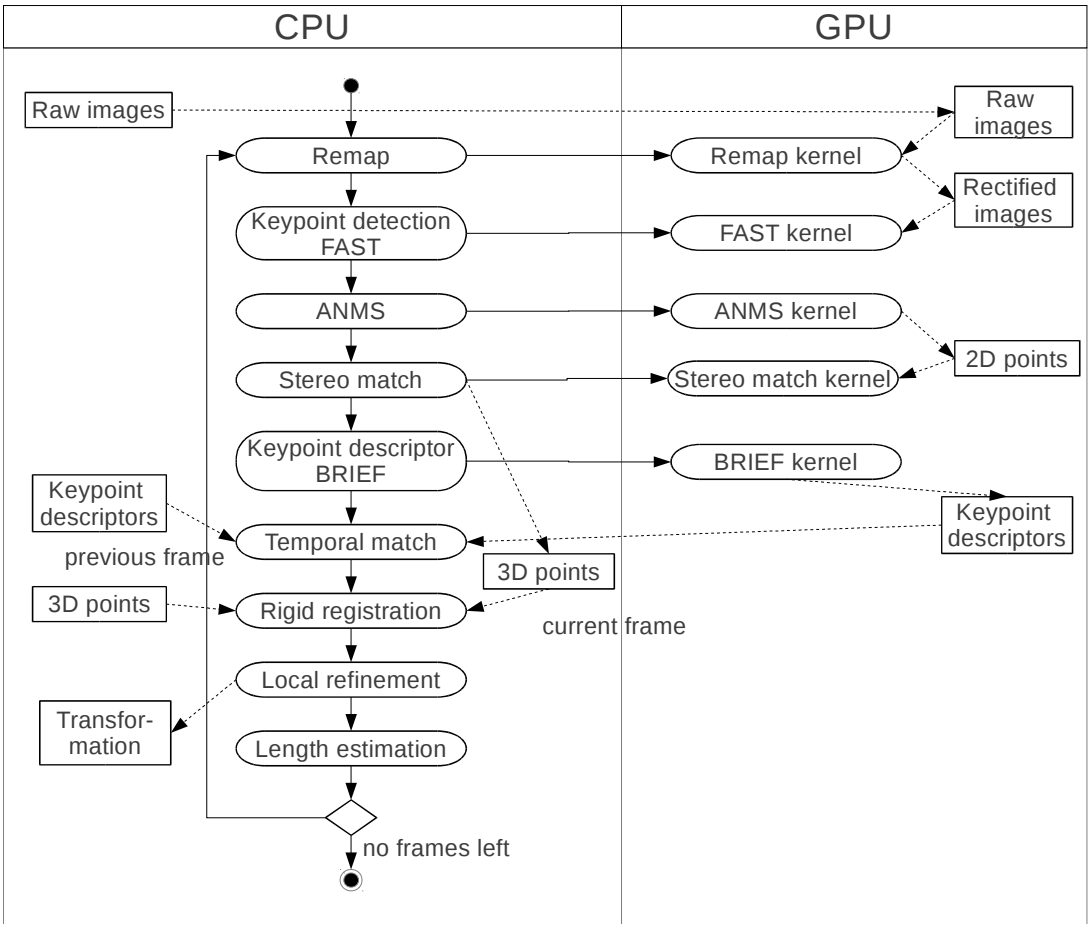


Illustration 40: UML activity diagram of GPU accelerated implementation.

It can be seen from the diagram that the current implementation requires some deep copies of the same structures: namely 2D keypoints and descriptors.

Asynchronous data transfer and CUDA streams could have been used to decrease response time. For example BRIEF descriptor computation could be started straight after FAST interest points would have been found. Descriptors are needed only in temporal match.

GPU acceleration was used neither in estimation of absolute orientation nor in local refinement. These steps were not a performance bottleneck.

4.5 Accuracy

Image 41 shows examples of accuracy of the length estimates applied on various tree types. Unfortunately implemented pipeline does not give estimates accurate enough. Perhaps used feature extractor is reason for inaccuracy. Earlier work used SURF as feature extractor. This thesis used FAST for its better performance. More work would be needed to find out how more accurate estimates could be achieved with real time performance.

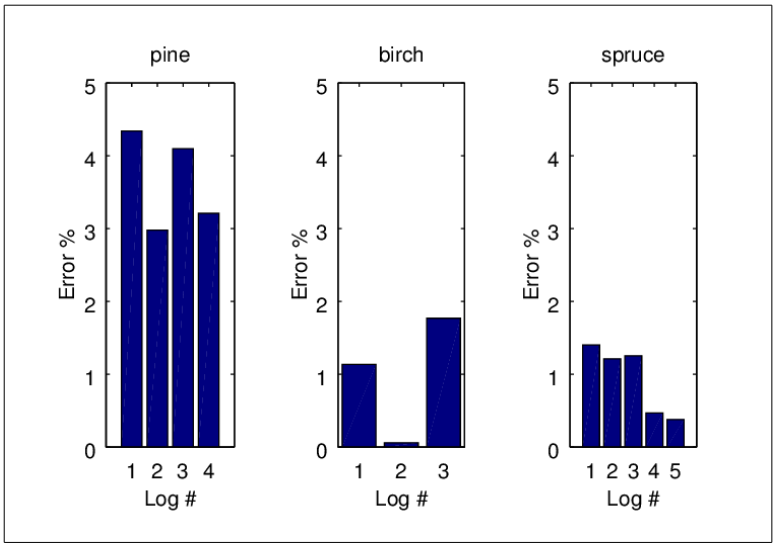


Illustration 41: Accuracy of estimates in selected trees.

Additionally it was found during development that image series for stereo calibration was inadequate. Feature points found in left and right images do not quite follow epipolar constraint. Sometimes this can be seen even by the naked eye. Histogram showing the error in y-direction of the keypoint position should be zero mean but this is not the case as can be seen in the image below.

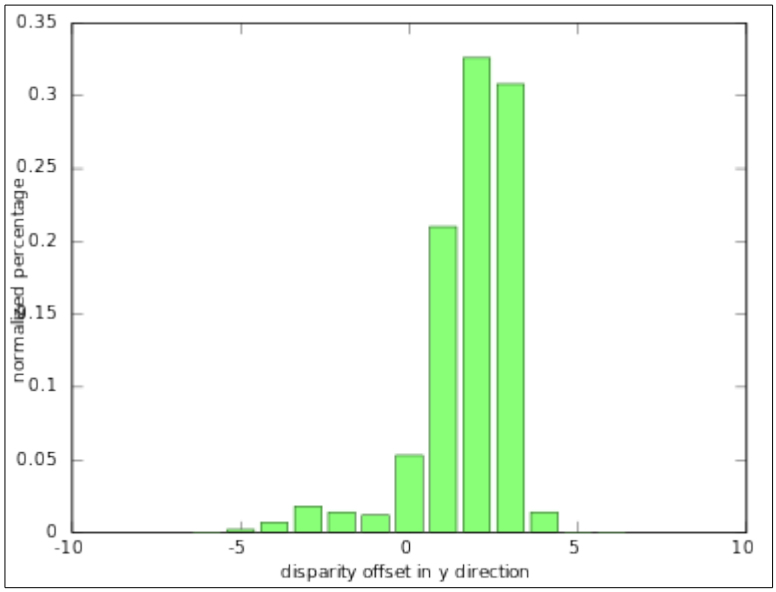


Illustration 42: Delta along y-axis should be zero mean under epipolar constraint.

5 Results

This section describes achieved performance. Log measurement was implemented on Ubuntu 12.04 LTS. Ubuntu is desktop operating system which does not guarantee even distribution of time on CPU threads. This gives some fluctuation in response times.

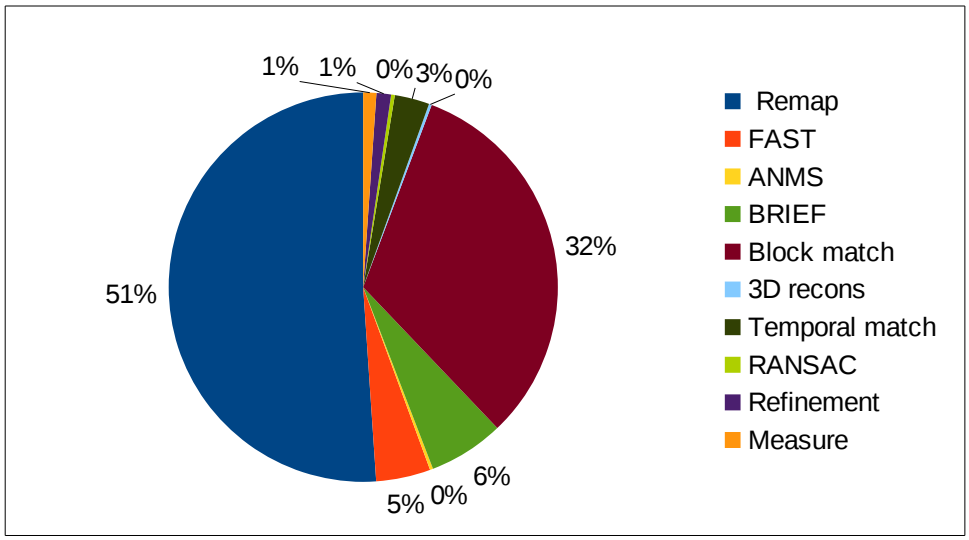


Illustration 43: Performance of single threaded measurement pipeline.

Image 43 shows performance of the single threaded pipeline. Performance is plotted as pie chart to demonstrate which subtasks are most time consuming. Two topmost are remap and block matching. 80 % of time is spent is these two subtasks. In these two subtasks possible speedup is most beneficial.

In following chapters subtasks are reviewed and gained speedup reported. However response time reviews start with memory allocations and transfer. Necessary drawback when using GPU co-processor.

5.1 Memory Allocations

Memory allocation in CPU RAM are shown in image 44. Up to about 10 MB allocation time appears to be almost constant. After that there is significant stepwise increase in allocation response. Allocated memory needs to be continuous. This is most probable reason why allocating big chunks of memory takes more time than smaller ones. Memory management needs to do more work when searching for the required amount of non-fragmented RAM. It is easier to find small amount of continuous memory, likely the first candidate will do.

Memory allocation in this thesis work do not exceed 10M bytes. CPU memory allocations are not going to be a problem with observed allocation times.

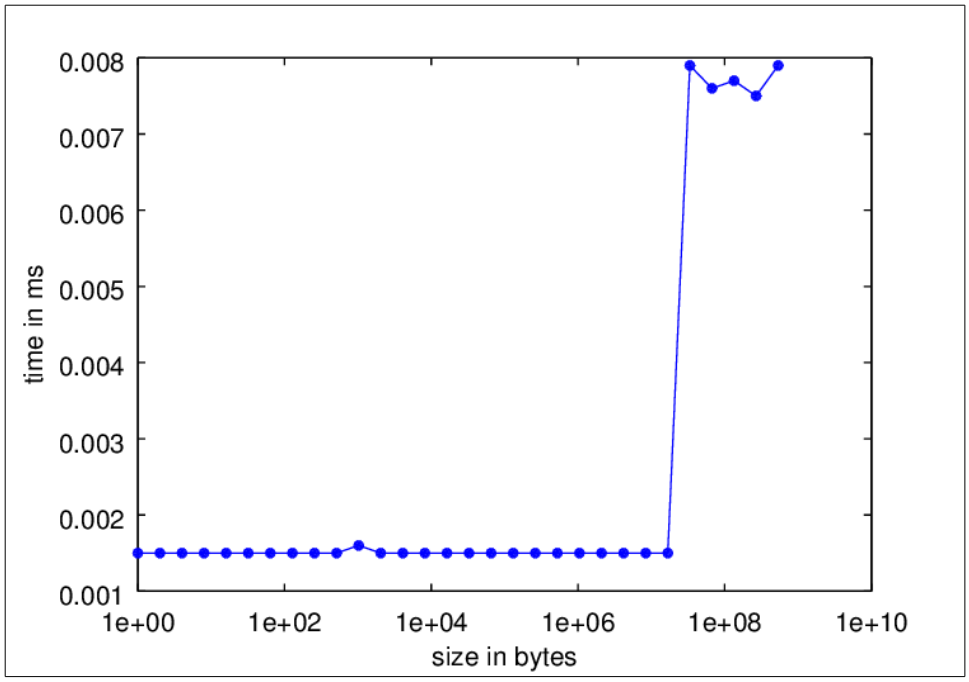


Illustration 44: Performance of CPU memory allocation.

Memory allocations in GPU global memory are presented in image 45. In GPU memory allocation response time appears to be almost constant up 10M bytes too. But time spent is 100 times longer. After 10 MB steep increase in allocation times appears.

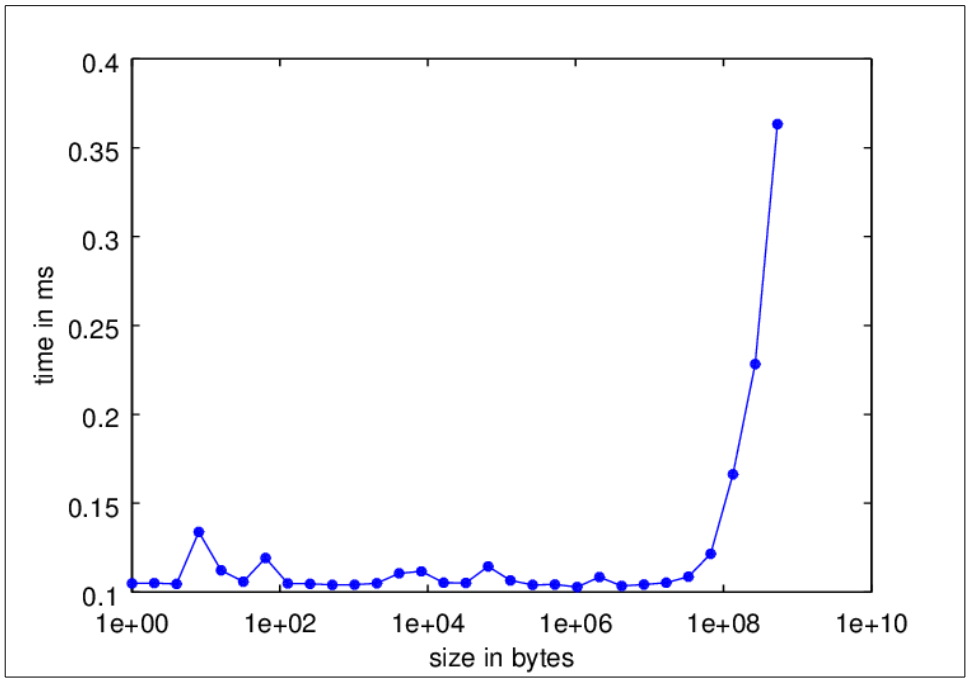


Illustration 45: Performance of GPU memory allocation.

Allocation in CPU RAM are about two order of magnitudes faster than GPU RAM. Ten separate GPU RAM allocations per frame add up to 1 ms, which is too much when little over 8 ms is the total allowed time. This means that preferred way to use GPU global memory is to preallocate

memory big enough and use same allocated lot over and over again. Freeing and reallocation would be a waste of time.

On the good side allocation time of GPU global memory keeps constant up to tens of megabytes.

In the given problem image size is 324 kB, so allocating memory takes about 0.1 ms.

5.2 Memory Transfer

Another drawback is the need to transfer data to GPU for processing and the other way round. Images are the biggest data size that need to be uploaded to the GPU but smaller entities are transferred as well: keypoints and descriptors.

Table 8 shows transfer rates of various PCI Express standards. Used hardware support PCI-E 16x.

Bus	Max bandwidth
PCI Express 1x	250 [500]* MB/s
PCI Express 2x	500 [1000]* MB/s
PCI Express 4x	1000 [2000]* MB/s
PCI Express 8x	2000 [4000]* MB/s
PCI Express 16x	4000 [8000]* MB/s
PCI Express 32x	8000 [16000]* MB/s

Table 8: Bandwidths defined by PCI-E standard.

Image 46 plots transfer times as a function of amount of data. Theoretical response time is also plotted in green. There seems to be constant overhead for small data transfers. Measured response converges with theoretical speed when the data size increases.

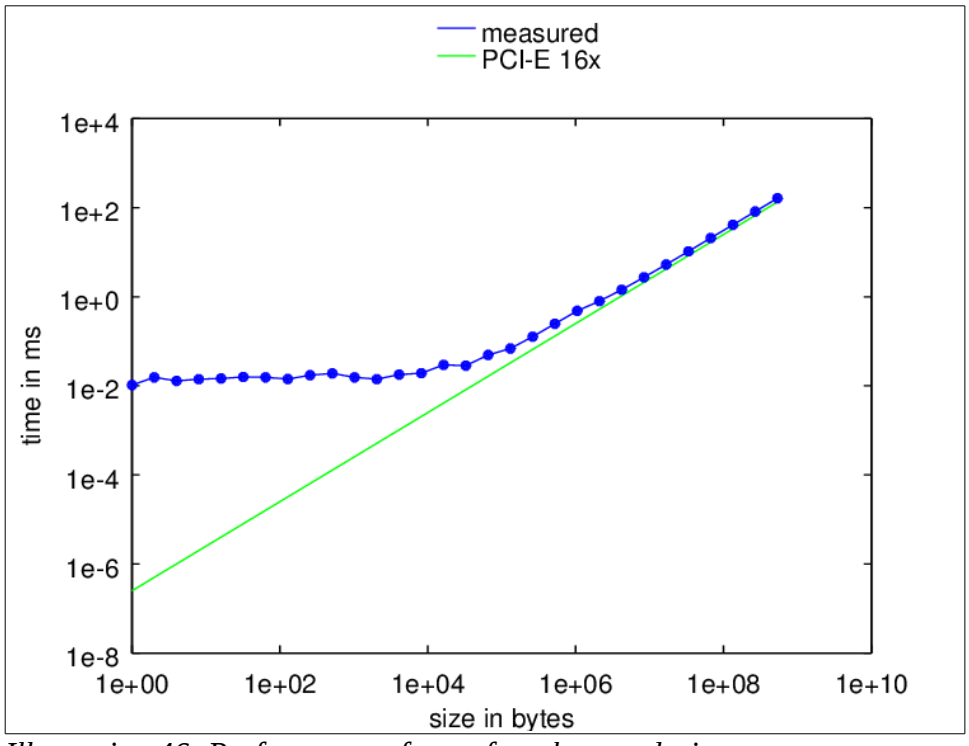


Illustration 46: Performance of copy from host to device.

Transfer time for stereo images used in this theses is around 0.2 ms. Response time of image transfer in this problem is not overwhelming. However, it can not be ignored. Multiple transfers between CPU and GPU do consume time.

5.3 Remapping

Remapping is an embarrassingly parallel process as value of each pixel can be calculated independently from others. More over when pixel values are read from the texture, interpolation is executed by hardware. However, naive implementation with global (device) memory performs almost as well.

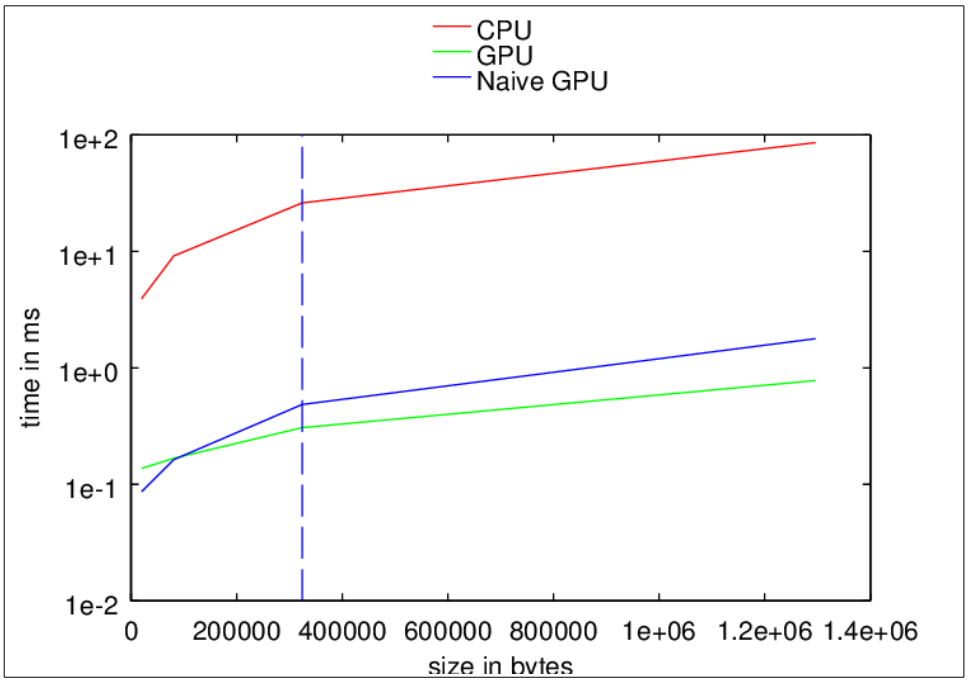


Illustration 47: Performance of remap

Image 47 shows performance of remap functions. Red is naive CPU implementation, blue is naive GPU implementation and green is GPU implementation with hardware texture interpolation. Vertical blue dash-line marks frame size used in this thesis. This gives 26/0.3=87 times speedup.

5.4 Keypoint detection FAST

Keypoint detection using FAST is also embarrassingly parallel. Each pixel can be tested independent of other pixels.

Image 48 shows the performance of different implementations of FAST keypoint detection. In this case, a simple bar chart was used. Performance of various implementations were measured keeping the image size fixed. “FAST CPU” gives a baseline for keypoint detection. This was single threaded implementation of FAST algorithm. “FAST GPU” is naive parallel version of FAST. Using texture improves performance as can be seen studying bar with label “FAST TEX GPU”. Improvement is explained by the fact that using texture lifts coalescing requirements. FAST detector reads global memory in a pattern that can't be coalesced. For each pixel, 16 pixels around it are read and they are not adjacent.

Bar with label “FAST OpenCV” gives equal good performance as best of three above. OpenCV uses MMX extensions in its CPU implementation. In this case it gave same performance as GPU accelerated version.

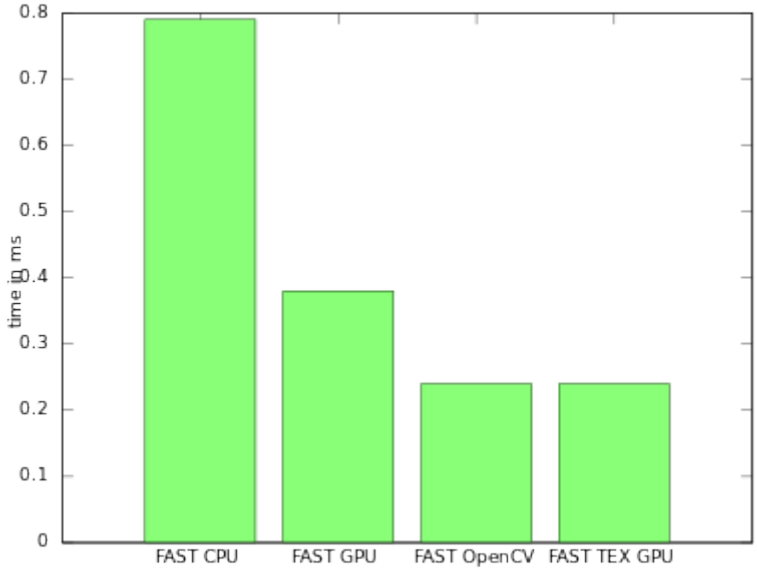


Illustration 48: Performance of FAST keypoint detection.

Why is speedup much less than in remap? As can be seen speed up is not even four. Kernel of FAST detection uses more registers than remap kernel. In fact with used hardware only one block could be executed by multiprocessor while remap executed 10 blocks simultaneously. This means that at best just 1024 threads are run in parallel but no more. Remember that CUDA had an upper limit for threads per block. Another difference is thread divergence. Remap included no if-statements while FAST includes a number of them.

Simple experimentation revealed that register pressure is the most probable reason for less speedup. Reducing the number of if-statements did not improve performance.

5.5 BRIEF descriptor

Calculating the BRIEF descriptor is an embarrassingly parallel problem as well. The descriptor can be calculated for each interest point independent of other interest points.

The following parallelization scheme was realized. For each keypoint a CUDA block was created. Number of threads in a block was equal to number of bytes in descriptor. So each thread was calculating one byte in the descriptor.

Performance of tested implementations are depicted in image 49. CPU version is single threaded implementation of BRIEF computation. Two GPU accelerated versions were implemented: with device memory and with texture. Texture gives slight decrease in response time – again.

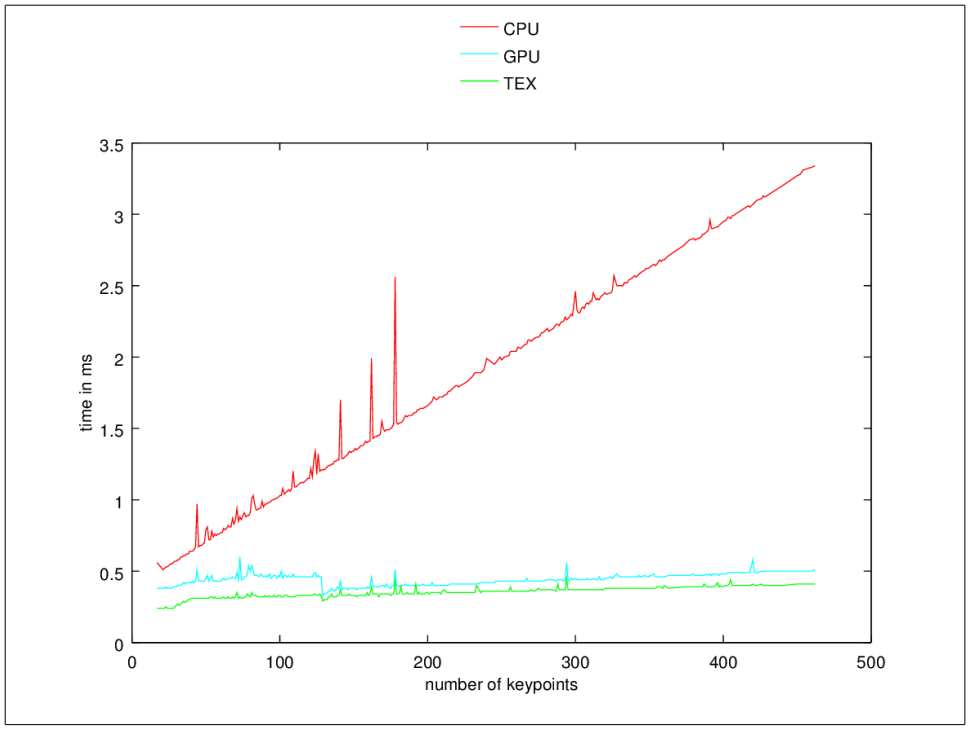


Illustration 49: Performance of BRIEF descriptor extractor.

Performance is limited by the number of concurrent blocks. Each multiprocessor can host at most 16 blocks, which is easily met. In used scheme number threads was equal to descriptor length in bytes: 16, 32 or 64. So at best there are 1024 (16*64) concurrent threads running on a multi-processor. There is not enough work available to fully utilize the GPU computing power.

GPU performance appears to be virtually constant time. This is really just virtual because the number of keypoints does not exceed the number of CUDA cores in used hardware. Performance would no longer appear constant if the number keypoints would be much larger than number of cores. Almost constant response time would be appealing if visual tracking would be used to control movement of log.

5.6 Nonmaximal suppression

Adaptive nonmaximal suppression is an embarrassingly parallel problem – again. Suppression radius for a keypoint can be calculated independently of suppression radii of other keypoints.

Single threaded CPU implementation was used as a baseline for other implementations. Suppression via Disc Covering (SDC) is also single threaded implementation but clearly faster than the naive algorithm. GPU implementation is faster than SDC version when number of keypoints is big. With few keypoints SDC turned out to be faster than GPU accelerated adaptive non-maximal suppression. Both implementations give significant improvement to the baseline, see image 50. Runtime complexity of simple implementation is quadratic where parallel implementation has almost constant time performance.

Here the simple scheme of a single block with 1024 threads was used. Each thread was assigned a keypoint. This thread computed the suppression radius for assigned keypoint. In the end, keypoints were sorted by suppression radius and k first returned as outcome. In sorting bitonic sort was used. This scheme gave a boost good enough. Especially when adaptive non-maximal suppression is not a

bottleneck of performance.

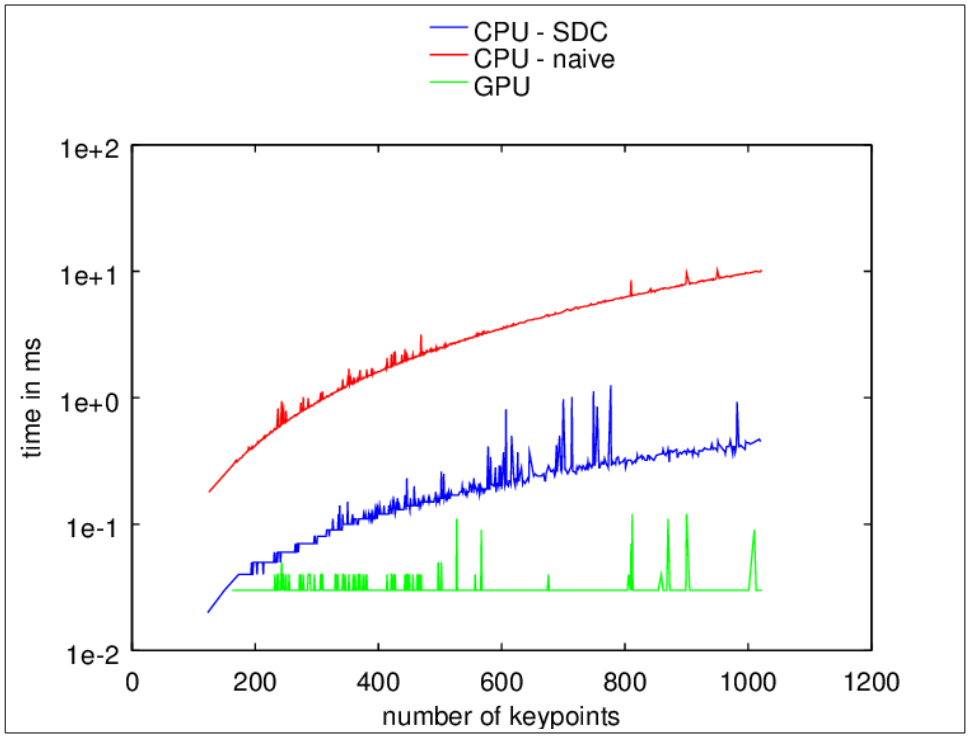


Illustration 50: Performance of adaptive non-maximal suppression presented. Note logarithmic scale in y-axis.

Parallelization of suppression via Disc Covering was also studied. CPU performance was greatly improved by using suppression via Disc Covering. However SDC turned out to be a hard algorithm to parallelize. It created dependencies between keypoints, and the problem was no longer embarrassingly parallel.

5.7 Stereo match

Stereo match was implemented by block match aka template match. Template matching consists of calculating the value of normalized correlation for each pixel in the search area. After that the maximum location is found by reduction.

Region of interest was calculated by CPU for each keypoint. After that a kernel to calculate normalized cross-correlation was launched. In the launch configuration each keypoint was assigned a thread block. The size of the thread block is equal to search area. Each location is assigned thread which calculates correlation value. Reduction is used to find location of best match after correlation values have been computed.

Only two implementations were used and measured: single threaded CPU and GPU. Image 51 plots performance of these implementations as a function of the number of keypoints. They both appear to be linear. CPU just has a steeper slope. This is quite expected as response per keypoint is the same. Number of the keypoints does not affect speedup. Number of the keypoints does affect overall performance by the speedup gained per keypoint.

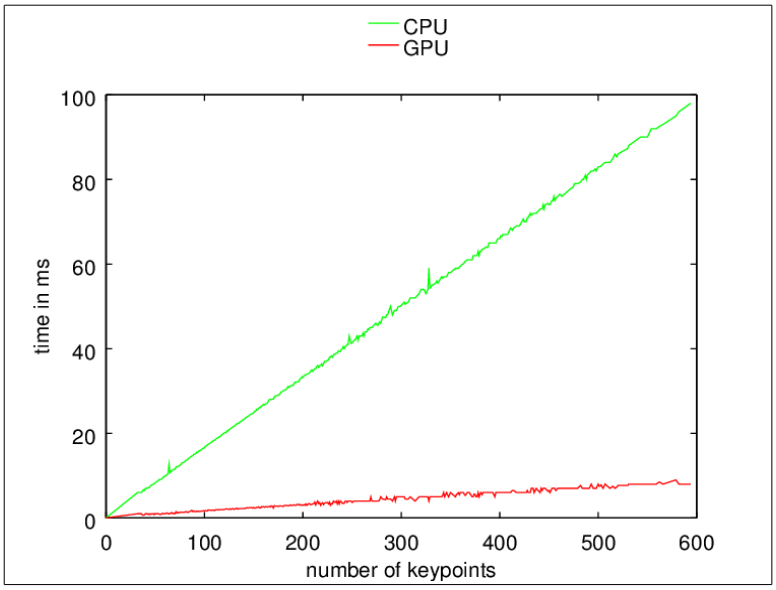


Illustration 51: Performance of template match as function of number of keypoints.

Image 52 plots performance as a function of block size. Performance plots appear no longer linear. The size of the template has a bigger effect on response in CPU implementation that in GPU implementation.

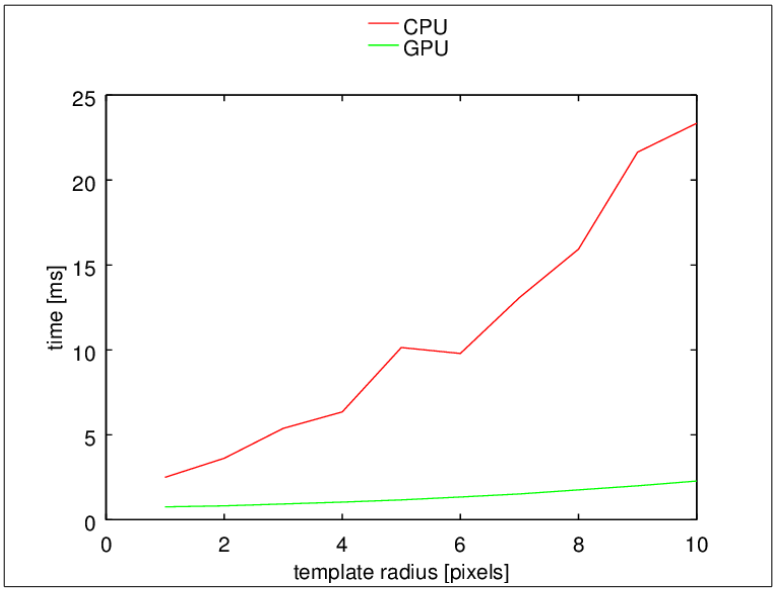


Illustration 52: Performance of template matching as function template size.

Profiling revealed that SM(X) can execute only one block at a time because of register pressure. This was quite a severe limitation as template matching was one of the two subtasks where most time was spent. Nevertheless, GPU gave some improvement over CPU.

5.8 3D reconstruction

3D reconstruction consists of match correcting and triangulation. Match correcting takes major part of response in 3D reconstruction. Almost all time in image 53 comes from match correcting.

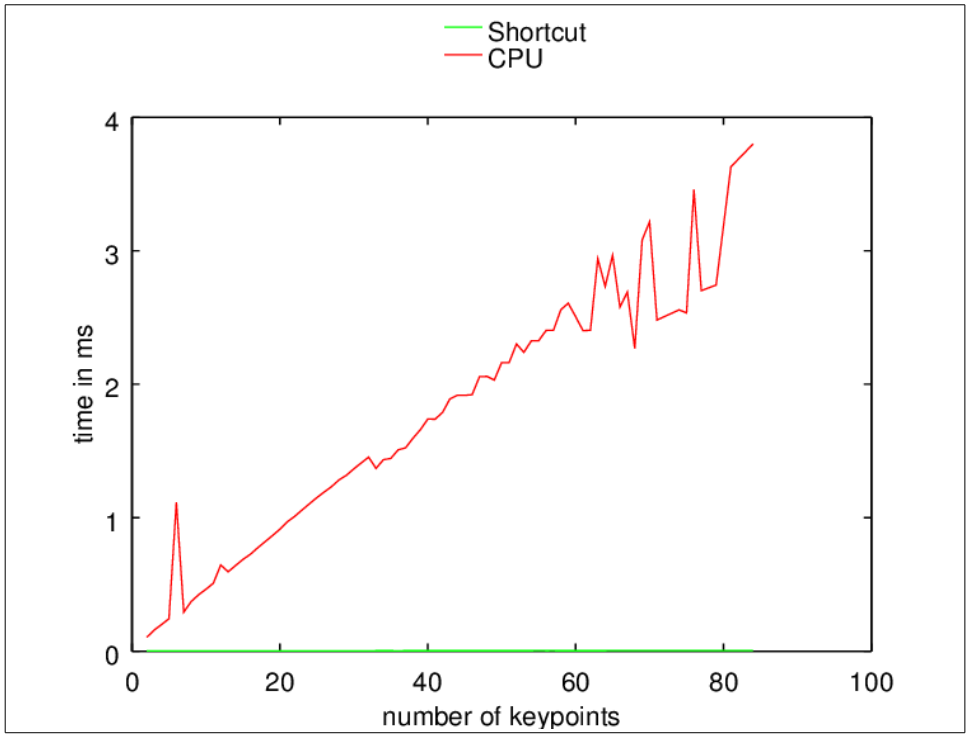


Illustration 53: Performance of 3D reconstruction.

To be precise most of time is spent in root finding for polynomials. Used root finding algorithm Durand-Kerner is easily parallelized. However performance was poorer than CPU version. There was only six roots to be found. GPU root finding would probable have beaten CPU if number of roots were higher.

Canonical stereo rig gave possibility to use shortcut in match correcting. Match correcting minimizes the geometrical error perpendicular to epipolar line. In canonical setting epipolar lines align with x-axis. So error can be expressed as follows

$$E = y^2 + (1-y)^2 = 2y^2 - 2y + 1$$
$$\frac{dE}{dy} = 4y - 2 = 0 \Rightarrow y = 0.5$$

Y-coordinate for corrected match can be found by simply taking average of y-coordinates of matched points.

5.9 Temporal match

Temporal match consists of Hamming distance calculation between descriptors and minimum finding. Computing Hamming distances is an embarrassingly parallel problem as distance for each keypoint in query set can be computed independent of other keypoints. Finding minimum is once again reduction problem.

Descriptor distances are computed into a two dimensional matrix. Size of matrix along x-axis is size

of query set. Size along second dimension is size of training set. This matrix contains distances between all descriptors. Value of each element in matrix is calculated by dedicated CUDA thread. If number of keypoints is around one hundred, size of matrix is around ten thousand. One thread-block can host 1024 thread at most. Distance calculation is divided into thread blocks of size 32x32. So grid size is obtained by dividing both dimensions by 32.

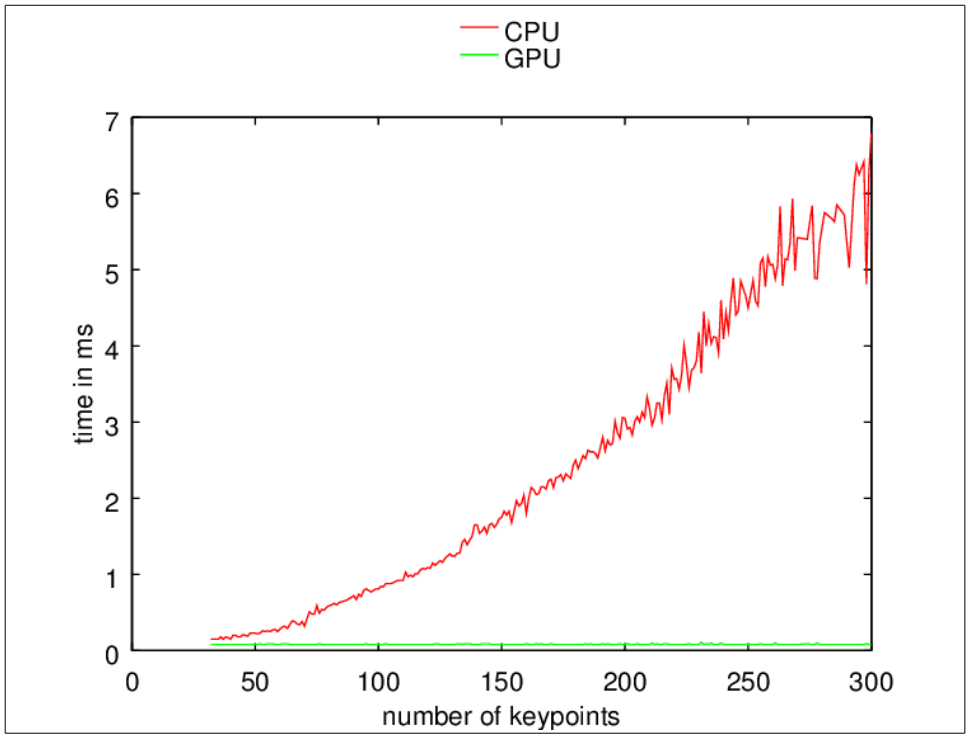


Illustration 54: Performance comparison of temporal match.

Image 54 plots performance of CPU and GPU versions of descriptor matching. Performance is plotted as a function of size of training set. Performance of GPU accelerated version keeps almost constant while performance of single threaded implementation becomes worse as training set increases.

Descriptors are loaded into shared memory to speed to distance calculation. Needed shared memory limits number of concurrent blocks per SM: three thread blocks can be assigned per multiprocessor in this implementation.

5.10 Rigid registration and local refinement

Rigid registration between reconstructed 3D points is found by help of RANSAC. Algorithm used withing RANSAC is absolute orientation.

Performance of the rigid registration is plotted in image 55. Some fluctuation is present as RANSAC is random by definition. In the used implementation the number of iterations is not fixed in advance.

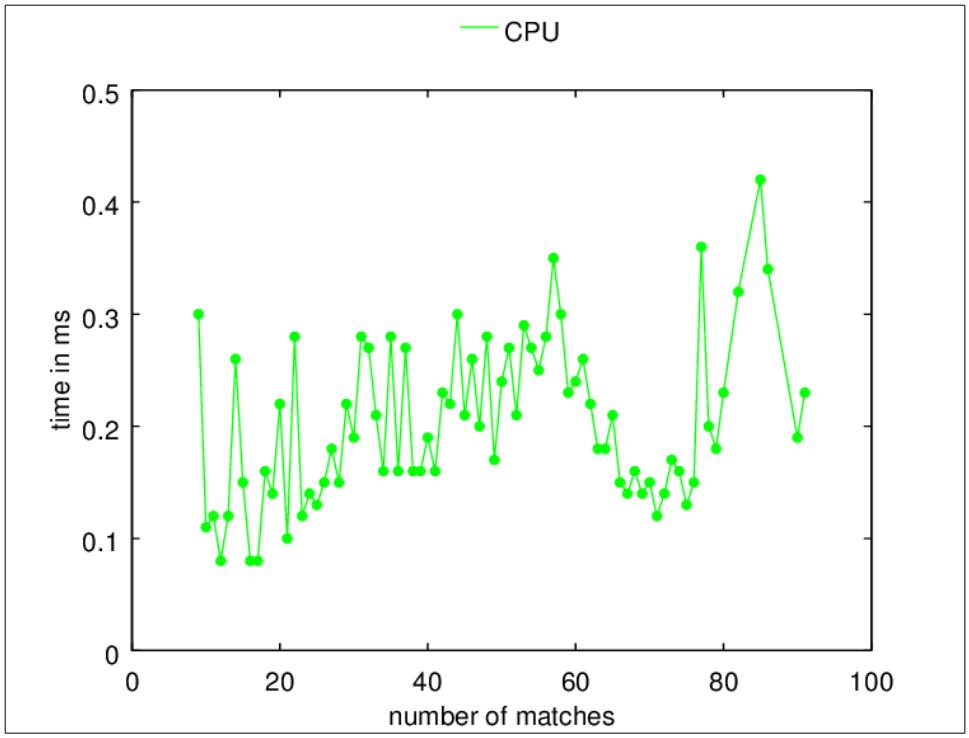


Illustration 55: Performance of rigid registration using RANSAC.

Parallel programming was not used in this step as RANSAC did not have significant impact in total performance. Two approaches could be used though. Inlier testing would be simple problem to parallelize. It would be embarrassingly parallel problem too. Another option would be parallelize by samples but this would be somewhat harder to implement.

This implementation included local refinement between last two frames. This is stripped down bundle adjustment: only two camera poses are considered. This being the last step in the pipeline, the number of keypoints being processed varies between 10 to 50.

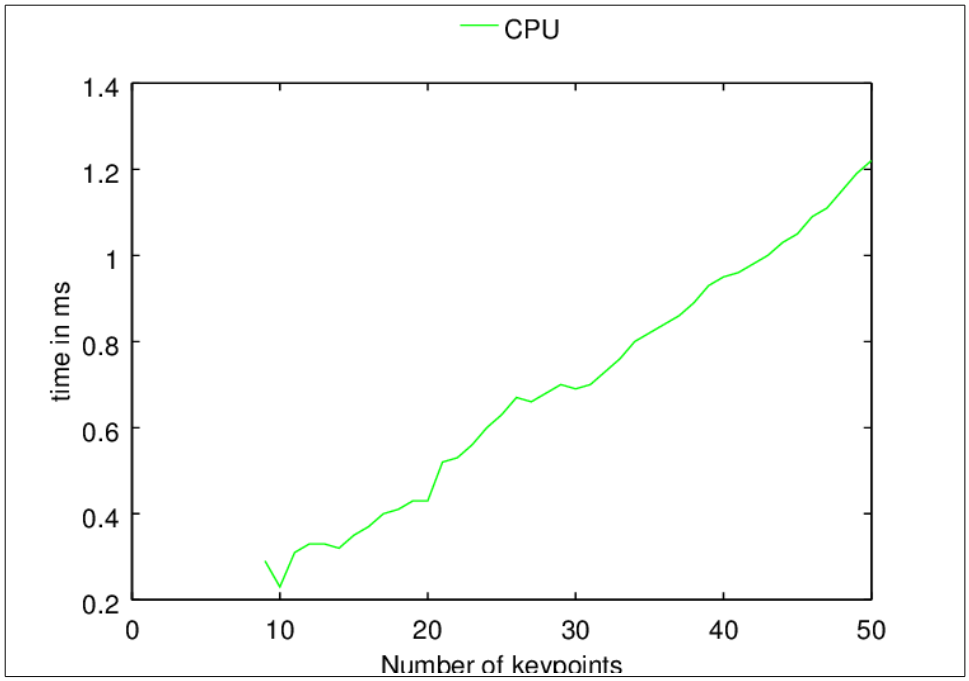


Illustration 56: Performance of single threaded local refinement.

Image 56 plots performance of local refinement. It can be easily noted that this would also be good candidate for parallelization. This could be achieved by using some linear algebra package including CUDA support. However, it remains unclear if the number of keypoints is too low to gain any speedup in execution time.

5.11 Performance of GPU

Now the subtasks has been reviewed and improvement has been found in them. Image 57 shows how the whole pipeline performs GPU accelerated. First, it can be noticed that response time is below required 8.3 ms. So this pipeline could be used at frame rate of 120Hz.

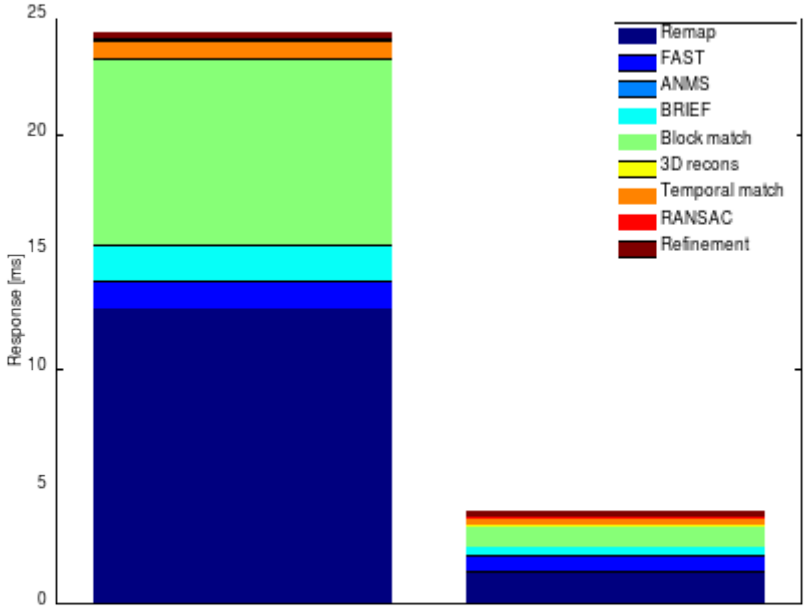


Illustration 57: Comparison of total performance.

Remap and block matching are still most time consuming subtasks. Most speedup could be gained by further improving performance of these two tasks.

Time used in copying data to or from GPU are included in subtasks which needed that data. Especially response time of remap includes also the transfer times of stereo images.

6 Summary and Recommendations

Real-time performance was achieved in this thesis work. First the algorithms in the subtasks were selected by performance. Second some of the algorithms were parallelized for speedup. Not all the algorithms needed speedup because single threaded CPU implementation had adequate performance.

Many of the subtasks were embarrassingly parallel algorithms, which promised good speedup in principle. However, physical constraints of CUDA capable graphics cards were limiting factor. For example register pressure in the block match turned out to be the limiting factor for parallelism. Shared memory was limiting factor in temporal match.

To summarize parallelization was not straightforward task but it required experimentation. Profiling tools gave important information regarding what is the bottleneck in parallelization.

Biggest speedup was achieved in remap. In this problem the number of needed threads was hundred of thousands, one thread for each pixel in rectified image. In later subtasks the natural element of the parallelization was a keypoint. The number of keypoints were few hundred at most. And later after RANSAC the support could be just few tens. It these algorithms the speedup was not so significant. It appears to be that CUDA helps most if elements of parallelization are ten of thousands or more.

Unexpected finding was that in some subtasks performance was virtually constant. Virtually constant time was found in following subtasks: adaptive non-maximal suppression, calculating BRIEF descriptor and temporal match. Probable reason for almost constant performance is that number of used CUDA threads was low compared to CUDA cores available. Constant response time would be really beneficial if the movement of the log would be controlled by software.

Further studying would be needed to resolve how needed accuracy would be reached while maintaining real time performance. This thesis came close to solution but not quite regarding accuracy. Perhaps accuracy was traded for performance is selecting algorithms for subtasks. However, findings in this thesis work would help the possible future work.

7 Bibliography

[1] M. Strandström, “Timber Harvesting and Long-distance Transportation of Roundwood 2013.” [Online]. Available: http://www.metsateho.fi/files/metsateho/Tuloskalvosarja/Tuloskalvosarja_2014_03b_Timber_harvesting_and_long-distance_transportation_of_roundwood_2013_ms.pdf. [Accessed: 15-Sep-2014].

[2] T. Melkas, “Wood measuring methods used in Finland 2012,” 2012. [Online]. Available: http://www.metsateho.fi/files/metsateho/Tuloskalvosarja/Tuloskalvosarja_2013_05b_Wood_measuring_methods_used_in_Finland_2012_tm.pdf. [Accessed: 27-Feb-2014].

[3] H. Strasdat, A. J. Davison, J. M. M. Montiel, and K. Konolige, “Double window optimisation for constant time visual SLAM,” in *2011 IEEE International Conference on Computer Vision (ICCV)*, 2011, pp. 2352–2359.

[4] A. F. Möbius, *Der barycentrische Calcül*. 1827.

[5] J. Plücker, *Analytisch-geometrische Entwicklungen*, vol. 2. 1831.

[6] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2004.

[7] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2011th ed. Springer, 2010.

[8] R. Baer and Mathematics, *Linear Algebra and Projective Geometry*. Mineola, N.Y: Dover Publications, 2005.

[9] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. Prentice Hall, 2011.

[10] G. H. van Golub, *Matrix Computations*, Fourth edition edition. Johns Hopkins University Press, 2012.

[11] D. Brown, “Decentering distortion of lenses (Lens decentering distortion using thin prism and Conrady models, noting application of precise calibration to analytical photogrammetry),” *Am. Soc. Photogramm. Annu. Conv. 31ST Wash. DC*, vol. 32, pp. 444–462, 1965.

[12] D. Brown, “Close range camera calibration,” *Photogramm. Eng.*, vol. 37, no. 8, p. 855, 1971.

[13] A. Conrady, “Decentered lens systems,” *Mon. Not. R. Astron. Soc. 79*, pp. 384–390, 1919.

[14] H. C. Longuet-Higgins, “A computer algorithm for reconstructing a scene from two projections,” *Nat. 1981 Vol2935828 P133*, vol. 293, no. 5828, p. 133, 1981.

[15] Q. T. Luong, “Matrice fondamentale et auto-calibration en vision par ordinateur,” University of Paris, Orsay, 1992.

[16] N. Ayache, “Rectification of images for binocular and trinocular stereovision,” *Pattern Recognit. 1988 9th Int. Conf. On*, pp. 11–16 vol.1, 1988.

[17] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*. Upper Saddle River, NJ: Prentice Hall, 1998.

[18] J. Heinly, E. Dunn, and J.-M. Frahm, “Comparative Evaluation of Binary Features,” in *Computer Vision – ECCV 2012*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds. Springer Berlin Heidelberg, 2012, pp. 759–773.

[19] E. Rosten and T. Drummond, “Fusing points and lines for high performance tracking,” in *Tenth IEEE International Conference on Computer Vision, 2005. ICCV 2005*, 2005, vol. 2, pp. 1508–1515.

[20] E. Rosten and T. Drummond, “Machine Learning for High-Speed Corner Detection,” in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Springer Berlin Heidelberg, 2006, pp. 430–443.

[21] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, “BRIEF: Binary Robust Independent

Elementary Features," in *Proceedings of the 11th European Conference on Computer Vision: Part IV*, Berlin, Heidelberg, 2010, pp. 778–792.
[22] M. Brown, R. Szeliski, and S. Winder, "Multi-image matching using multi-scale oriented patches," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005*, 2005, vol. 1, pp. 510–517.
[23] A. Behrens and H. Röllinger, "Analysis of feature point distributions for fast image mosaicking algorithms," *Acta Polytech. J Adv. Eng.*, vol. 50, no. 4, pp. 12–18, 2010.
[24] S. Gauglitz, L. Foschini, M. Turk, and T. Hollerer, "Efficiently selecting spatially distributed keypoints for visual tracking," in *2011 18th IEEE International Conference on Image Processing (ICIP)*, 2011, pp. 1869–1872.
[25] B. Cyganek and J. P. Siebert, *An Introduction to 3D Computer Vision Techniques and Algorithms*, 1 edition. Chichester, U.K: Wiley, 2009.
[26] B. Pan, H. Xie, and Z. Wang, "Equivalence of digital image correlation criteria for pattern matching," *Appl. Opt.*, vol. 49, no. 28, pp. 5501–5509, Oct. 2010.
[27] D. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," *Fourth Int. Conf. Comput. Vis. Theory Appl.*, 2009.
[28] N. Sünderhauf and P. Protzel, "Stereo Odometry - A Review Of Approaches," Chemnitz University of Technology, Technical report, 2007.
[29] "Navigation using affine structure from motion - Springer." [Online]. Available: http://link.springer.com.libproxy.aalto.fi/chapter/10.1007/BFb0028337. [Accessed: 05-Jan-2015].
[30] R. Hartley, "Stereo from uncalibrated cameras," *Comput. Vis. Pattern Recognit. 1992 Proc. CVPR 92 1992 IEEE Comput. Soc. Conf. On*, pp. 761–764, 1992.
[31] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 edition. Cambridge, UK ; New York: Cambridge University Press, 2007.
[32] R. Hartley and P. Sturm, *Triangulation*. 1996.
[33] É. Durand, *Solutions numériques des équations algébriques: par E. Durand,.... Équations du type F (x)*. Masson et Cie, 1960.
[34] I. O. Kerner, "Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen," *Numer. Math.*, vol. 8, no. 3, pp. 290–294, May 1966.
[35] K. Kanatani, Y. Sugaya, and H. Niitsuma, "Triangulation from two views revisited: Hartley-Sturm vs. optimal correction," in *In: Proc. 19th British*, 2008, pp. 173–182.
[36] Martin A. Fischler, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, 1981.
[37] R. I. Hartley, *Projective Reconstruction and Invariants from Multiple Images*. 1994.
[38] P. H. S. Torr and D. W. Murray, "The Development and Comparison of Robust Methods for Estimating the Fundamental Matrix," *Int. J. Comput. Vis.*, vol. 24, no. 3, pp. 271–300, Sep. 1997.
[39] B. K. P. Horn, "Closed-form solution of absolute orientation using unit quaternions," *J. Opt. Soc. Am. A*, vol. 4, no. 4, pp. 629–642, 1987.
[40] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-Squares Fitting of Two 3-D Point Sets," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-9, no. 5, pp. 698–700, Sep. 1987.
[41] P. J. Besl and N. D. McKay, "A method for registration of 3-D shapes," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 2, pp. 239–256, Feb. 1992.
[42] Y. Chen and G. Medioni, "Object modeling by registration of multiple range images," in *IEEE International Conference on Robotics and Automation.*, 1991, vol. 3, pp. 2724–2729.

[43] P. H. Schönemann, “A generalized solution of the orthogonal procrustes problem,” *Psychometrika*, vol. 31, no. 1, pp. 1–10, Mar. 1966.
[44] D. C. Brown, “A solution to the general problem of multiple station analytical stereo triangulation,” Patrick Airforce Base, Florida, Technical Report RCA-MT Data Reduction Technical Report No 43, 1958.
[45] D. C. Brown, “The bundle adjustment - progress and prospects.,” *Int Arch. Photogramm.*, vol. 21(3), p. 33, 1976.
[46] H. Strasdat, *Local accuracy and global consistency for efficient SLAM*. Imperial College London, 2012.
[47] C. . Harris, “3D positional integration from image sequences,” *Image Vis. Comput.*, vol. 6, no. 2, p. 87, 1988.
[48] Noah Snavely, “Photo tourism: exploring photo collections in 3D,” *SIGGRAPH 2006 Pap.*, Jul. 2006.
[49] H. Moravec, “Obstacle avoidance and navigation in the real world by a seeing robot rover [Ph.D. Thesis],” p. 177, 1980.
[50] S. Lacroix, “Rover self localization in planetary-like environments,” *ISAIRAS99 Proc. Fifth Int. Symp. Artif. Intell. Robot. Autom. Space Noordwijk Neth. Int. Organ.*, pp. 433–440, 1999.
[51] C. F. Olson, “Robust stereo ego-motion for long distance navigation,” *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, vol. 2, pp. 453–458, 2000.
[52] D. Nister, “Visual odometry,” *Proc. 2004 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. 2004*, vol. 1, pp. I–I, 2004.
[53] J. A. Hesch, “A Direct Least-Squares (DLS) method for PnP,” *2011 Int. Conf. Comput. Vis.*, pp. 383–390, 2011.
[54] X.-S. Gao, “Complete solution classification for the Perspective-Three-Point problem.(Author Abstract),” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 8, p. 930, 2003.
[55] D. Nister, “An efficient solution to the five-point relative pose problem,” *2003 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2, pp. II–195, 2003.
[56] “NVIDIA Kepler GK110 Architecture Whitepaper.docx - NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.” [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. [Accessed: 11-Nov-2014].
[57] “Programming Guide :: CUDA Toolkit Documentation.” [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy__grid-of-thread-blocks. [Accessed: 10-Nov-2014].
[58] V. Volkov and J. Demmel, “LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs,” EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2008-49, May 2008.
[59] V. Volkov, “Benchmarking GPUs to tune dense linear algebra,” *High Perform. Comput. Netw. Storage Anal. 2008*, pp. 1–11, 2008.
[60] “Best Practices Guide :: CUDA Toolkit Documentation.” [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#arithmetic-instructions. [Accessed: 28-Jan-2015].
[61] R. Farber, *CUDA Application Design and Development*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.
[62] M. Harris, “Optimizing Parallel Reduction in CUDA.” [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. [Accessed: 25-Nov-2014].
[63] “Faster Parallel Reductions on Kepler | Parallel Forall.” [Online]. Available:

http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/. [Accessed: 17-Nov-2014].
[64] N. Wilt, *CUDA Handbook: A Comprehensive Guide to GPU Programming, The*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2013.
[65] K. E. Batcher, “Sorting Networks and their Applications,” *1968 Spring Jt. Comput. Conf.*, vol. 32, no. AFIPS Proc, pp. 307–314, 1968.
[66] “GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics,” *GPU Gems Program. Tech. Tips Tricks Real-Time Graph.*, 2004.
[67] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st edition. Sebastopol, CA: O’Reilly Media, 2008.
[68] R. Duraiswami, “Memory management for performance.” [Online]. Available: http://www.umiacs.umd.edu/~ramani/cmsc828e_gpusci/Lecture7.pdf. [Accessed: 27-Nov-2014].